Universidad Carlos III de Madrid

Escuela Politécnica Superior

Grado en Ingeniería Informática

ANÁLISIS Y DESARROLLO DE UN PROGRAMA DE AJEDREZ

Autor: Roberto Campos Ortiz

Tutor: Carlos Linares López

En su grave rincón, los jugadores
rigen las lentas piezas. El tablero
los demora hasta el alba en su severo
ámbito en que se odian dos colores.

[…]

Tenue rey, sesgo alfil, encarnizada
reina, torre directa y peón ladino
sobre lo negro y blanco del camino
buscan y libran su batalla armada.

[…]

También el jugador es prisionero
(la sentencia es de Omar) de otro tablero
de negras noches y de blancos días.

Dios mueve al jugador, y éste, la pieza.
¿Qué Dios detrás de Dios la trama empieza
de polvo y tiempo y sueño y agonía?

Ajedrez, Jorge Luis Borges

Agradecimientos

Quiero agradecer su apoyo a todas aquellas personas, que de una forma u otra han hecho posible que yo haya podido realizar este proyecto.

Especialmente a mis padres, Julio y Cory, en general por tantas cosas que no caben en este agradecimiento, y en especial, por estar siempre en el camino para apoyarme en todo momento.

A Ángeles, por la ayuda y el cariño que ha puesto en este proyecto a través de mí. Por haber sabido, además, escuchar, aconsejar, apoyar, y tantas cosas más que sólo ella es capaz de hacer.

A Carlos, mi tutor, por haber gastado tantas fuerzas en darme ánimos en todo momento, y haberme enseñado tanto durante este proyecto.

Índice General

Índice de ilustraciones

Índice de tablas

Capítulo 1.

Introducción

Una vez que los ordenadores han ido cubriendo las funcionalidades iniciales para las que fueron creados (reducción de los tiempos de cómputo, almacenamiento de datos, comunicaciones,…), comenzaron a abrirse nuevas líneas de investigación, cuyo objetivo era que estos dispositivos resolviesen problemas, cada vez más complejos, de forma autónoma.

Las mejoras en las capacidades técnicas de los equipos (innovaciones hardware), y el nuevo software que fue surgiendo, posibilitaron manejar un mayor volumen de datos a una velocidad mucho mayor. Todo ello, provocó que cambiase de manera radical la forma en la que se veía la informática, pasando de ser una ciencia sólo aplicada en la ingeniería a convertirse en una herramienta para cualquier disciplina.

En este punto, surge la idea de, tratar de desarrollar software que sea capaz de resolver un problema o tomar decisiones por sí mismo. Debido a las mejoras contadas anteriormente, estos programas tienen la posibilidad de hacer más operaciones por segundos, y de almacenar estas operaciones en memoria intermedia para aumentar la velocidad de computación.

En este contexto surge la Inteligencia Artificial (IA), con el fin de simular el razonamiento y aprendizaje humano en un ordenador. Sin embargo, esta IA puede ser abordada desde distintas perspectivas. Se puede tratar de conseguir que un ordenador “razone” como un ser humano, pero con las características “aumentadas” de su hardware (mayor capacidad de cálculo, procesamiento y almacenamiento de datos,…), en esta corriente encontraríamos los algoritmos evolutivos, las redes de neuronas o la robótica humanoide. Otra forma de enfocar la IA muy diferente, es entender que es difícil emular el cerebro

humano, debido principalmente a su complejidad y a que todavía no se conoce su funcionamiento exacto, y por ello, es posible desarrollar una IA que sólo explote las amplias capacidades en las que un ordenador supera a un humano (ya comentadas). Siendo el programador, a través de la definición de ciertas reglas, quién guie a la Inteligencia Artificial. Por ejemplo, a través de las heurísticas, es el humano quien decide lo que es inteligente o no, es decir, indica al programa el patrón a seguir para alcanzar el objetivo, y aprovecha, por ejemplo, la capacidad del hardware de evaluar decisiones a una gran profundidad.

Además, el desarrollo de la IA presenta numerosos desafíos técnicos, ya que necesita realizar muchas operaciones que consumen cuantiosos recursos. Esto provoca que no exista una forma clara de afrontar el desarrollo de un programa que implemente IA, y sea difícil determinar si este programa es eficiente o no en el manejo de los recursos. Por ello, existen numerosas estrategias a la hora de desarrollar un algoritmo, siendo cada una de ellas adecuada al problema para el que se apliquen.

Hoy en día, la IA es aplicada en campos como [CITIC, 2014] la robótica (en la que trata de dar autonomía a un robot para que realice una tarea no concreta y repetitiva), medicina (monitorización inteligente de pacientes, detección de patrones clínicos en señales biomédicas,…), economía (análisis del fracaso empresarial, estimación de riesgos y rentabilidad de productos financieros,…), ingeniería industrial (mantenimiento predictivo de componentes mecánicos, diseño de diques verticales,…) y, por supuesto, en la informática, donde podemos destacar, los algoritmos evolutivos, las redes de neuronas o las

heurísticas aplicadas. También el campo de la informática, se aplica la IA a todo tipo de juegos, para encontrar de la forma más eficiente posible la solución. Por ejemplo, para juegos de dominio finito y discreto, N-puzzle o Rush hour; para juegos de estrategia, Risk, ajedrez o Go; para videojuegos ya sean shooters, aventuras gráficas, de rol, etc.

Este proyecto, se centra en la realización de programa que juegue al ajedrez basándose en una heurística y apoyándose en un árbol de búsqueda, para encontrar en cada movimiento la mejor jugada según la heurística definida en el menor tiempo posible.

El ajedrez es un juego de estrategia, en el cual se enfrentan dos jugadores, que tienen a su disposición distintas piezas para moverlas por un tablero. El juego de piezas está compuesto por 16 trebejos que pueden moverse con restricciones a lo largo de un tablero de 64 casillas, por lo que las posibles combinaciones de movimientos son prácticamente infinitas. El origen del ajedrez es desconocido, pero sí se puede asegurar que es realmente antiguo, ya que en excavaciones efectuadas en Mesopotamia se han encontrado objetos que demostraban que el ajedrez o un juego similar existía por lo menos 4000 años antes de Cristo[Fénix, 2012]. Esto hace, que tanto por su historia como por su estructura de juego, el ajedrez sea un juego de estrategia conocido y jugado mundialmente, que además incita a desplegar sobre su juego todas las técnicas conocidas de IA. En 1996, IBM, desarrolló una supercomputadora llamada Deep Blue, que llegó a ganar en tres ocasiones al, por aquel entonces campeón del mundo de ajedrez, Gary Kasparov. La estrategia de Deep Blue, era aprender de los movimientos de grandes ajedrecistas para jugar mejor.

En este proyecto, se trata de crear un jugador de ajedrez que no aprenda movimientos, sino que simplemente se guíe, desde el principio por una heurística previamente definida. El desarrollo de este proyecto, se explica detalladamente en los siguientes capítulos de esta memoria, que están organizados de esta manera:

Capítulo 2 - Estado de la cuestión: En este apartado se detalla, cómo funcionan las heurísticas aplicadas a juegos de estrategia como el ajedrez, así como los distintos árboles de búsqueda que se pueden aplicar a estar heurísticas, explicando las ventajas y los inconvenientes de cada uno de ellos. Además, se explican brevemente algunos de los aspectos más importantes del ajedrez a tener en cuenta para realizar el proyecto.

Capítulo 3 - Objetivos: En este apartado, se definen y explican brevemente los objetivos perseguidos al realizar el proyecto.

Capítulo 4 - Desarrollo: En este cuarto capítulo, se exponen y detallan todos los pasos realizados durante el desarrollo del proyecto. Empezando por explicar el diseño e implementación del tablero, la adaptación de la heurística y el esquema del árbol de búsqueda creado para el proyecto.

Capítulo 5 - Pruebas y resultados: Una vez terminado el proyecto, se probará en distintos en tornos, para ver los resultados obtenidos.

Capítulo 6 - Conclusiones: En este apartado, se chequea si el proyecto ha alcanzado los objetivos fijados, y las conclusiones obtenidas durante la realización del proyecto.

Capítulo 7 - Líneas futuras: En este capítulo, se explica cuáles podrían ser las siguientes evoluciones del proyecto realizado.

Anexo A - Costes del proyecto: Se hace un cálculo de los costes de implementación del proyecto.

Anexo B - Bibliografía: Se referencia toda la bibliografía citada durante la memoria.

Anexo C - Explicación en inglés del proyecto: En este capítulo se explican, en inglés, los principales puntos del proyecto llevado a cabo.

Capítulo 2.

Estado de la cuestión

Los juegos de estrategia siempre son atractivos para el ser humano, debido al reto mental que plantean. Por ello, han sido objeto de múltiples análisis y fuente de inspiración para científicos. Con el desarrollo de los ordenadores, cabía esperar que surgiera un interés creciente en desarrollar algoritmos capaces de jugar a estos juegos de estrategia.

Por su parte, el ajedrez, es probablemente, el paradigma de estos juegos de estrategia. Jugado desde la antigüedad combina sencillez en las normas con un infinito abanico de movimientos, lo que provoca que no existan dos partidas iguales.

En este proyecto se trata de crear un jugador de ajedrez. Como ya se ha comentado anteriormente, hay distintas formas de enfocar este problema. También hemos visto que en su momento, Deep Blue, consiguió aprender movimientos de humanos para después derrotar a humanos. Con Deep Blue, se trató de emular el sistema de aprendizaje del cerebro humano para que una máquina aprendiese a jugar. También se intentó de simular el sistema de razonamiento de un humano para que con los conocimientos previamente adquiridos la máquina pudiese de forma razonada generar movimientos que le llevasen a ganar la partida. Como se vio en su momento, este enfoque del problema llegó a ser eficaz, ya que Deep Blue ganó al campeón del mundo en aquel momento [Campbell, 2002]. Sin embargo, la cuestión es si esa máquina realmente era eficiente, y consiguió resolver el problema de la mejor manera posible.

Deep Blue, aprovechó los aspectos en los que un ordenador supera a un humano para almacenar y analizar casos de partidas ya existentes, y entre ellas elegir la

opción que consideraba más adecuada. Sin embargo, hay otra forma de afrontar el problema. Si el jugador de ajedrez que se desarrolla, en vez de utilizar sus recursos para recordar y elegir partidas, los usa para, siguiendo unas reglas que el programador le ha fijado, analizar todos los posibles casos (o al menos los que siguiendo un árbol de búsqueda parezcan prometedores) a gran profundidad, puede conseguir mejores resultando utilizando menos recursos. El principal inconveniente de esta forma de afrontar el problema, es que las reglas definidas por el programador no tienen por qué ser óptimas para el problema que se tratar de resolver. Precisamente, este es uno de los mayores retos del programador, saber modelar la realidad en función a unas reglas heurísticas que ayuden al ordenador a seguir una estrategia óptima cuando se enfrente a un problema.

2.1 Ajedrez

El ajedrez es un juego de estados finitos, en el que dos jugadores se enfrentan para tratar de batirse el uno al otro. Por ello, es denominado según la teoría de juegos, como un juego de suma cero, en el que el beneficio total de los jugadores suma cero, ya que se gana exactamente la misma cantidad que pierde el oponente. Este es un análisis del ajedrez es bastante importante, porque nos permite darnos cuenta de cómo se debe diseñar la estrategia del juego.

Además, es necesario analizar cómo se desarrolla una partida de ajedrez, según van realizando los movimientos cada uno de los jugadores. El ajedrez consta de tres fases de juego, analizadas a continuación.

2.1.1 Fases del ajedrez

Las fases en las que podemos dividir el ajedrez son: aperturas, medio juego y final del juego. Esta diferenciación, resulta de gran ayuda al analizar una partida de ajedrez, ya que cada fase, tiene sus propias características, y por ende, una estrategia propia.

2.1.1.1 Aperturas

Las aperturas en ajedrez se corresponden con la primera fase del juego, y suponen una parte realmente importante de este. Durante la fase de apertura de la partida que suele durar entre 5 y 15 movimientos, se trata de desarrollar todas las piezas menores que bloquean el camino al resto de piezas. Durante esta fase también es importante tratar de controlar cierta zona de casilla para desarrollar desde estas casillas la estrategia planteada. Generalmente se trata de capturas las casillas centrales desde las cuales se puede controlar más fácilmente el resto del tablero. Los objetivos principales de una apertura son:

- Desarrollar las piezas menores
- Control del centro del tablero
- Mantener la seguridad del rey, generalmente con un enroque (corto o largo)
- Formar una estructura de peones

Existen miles de aperturas desarrolladas por los jugadores durante los muchos años de existencia del ajedrez, aunque existe una clasificación de las aperturas más generales, y luego se van clasificando las distintas variantes de estas aperturas, existen miles de variantes dentro de una misma apertura, por lo que las combinaciones una vez pueden ser infinitas.

Los Grandes Maestros de ajedrez llegan a tener la capacidad de memorizar cientos de aperturas, así cuando un oponente comience siguiendo una apertura ellos la reconocerán rápidamente y sabrán casi de forma automática que posibilidades tienen. Además, no perderán mucho tiempo en realizar los movimientos automáticos ya que sólo tendrán que seguir las aperturas previamente aprendidas.

Con relación a un programa que juegue al ajedrez, existen varias ventajas en tener almacenadas estas aperturas, frente a tener que calcular movimientos mediantes IA. Primero, la ventaja más evidente, es el ahorro de tiempo que supone, simplemente buscar el movimiento que sucede al recientemente realizado, frente a tener que calcular el siguiente movimiento con IA (llamadas a la función de evaluación y búsqueda mediante el algoritmo de búsqueda). Este ahorro de tiempo supone ejecutar muchos movimientos rápidos, con una estrategia definida y sin riesgo de caer en trampas. Este tipo de comportamiento se asemeja mucho a una partida clásica entre humanos donde se ve como los primeros turnos transcurren a una alta velocidad.

La segunda gran ventaja de utilizar aperturas, es que no es necesario tener en cuenta el efecto horizonte, ni existe la posibilidad de caer en una trampa, debido a que son aperturas realizadas por otros jugadores, en las que ya se sabe cuál ha

sido el resultado final de la partida, por lo que se sabe si es o no conveniente utilizar esa apertura en ese caso. Aunque esta situación parezca no ser tan relevante, es algo de vital importancia a la hora de desarrollar una estrategia durante la partida, y llegar al medio juego con la partida igualada o incluso con una ligera ventaja. Además, esto ayuda a la agente de IA a calcular de forma más precisa los siguientes movimientos.

2.1.1.2 Medio juego

Esta es la fase del juego que generalmente, suele ser la más extensa, donde se producen más movimientos. Hasta ella, los jugadores llegan habiendo realizado pequeñas estrategias durante las aperturas, que han podido o no terminar consolidándose. La situación más normal, es que uno de los jugadores, llegue con una ligera ventaja sobre el oponente, lo cual le permitirá durante esta fase consolidar o desarrollar su propia estrategia, llevando la iniciativa de la partida. Por el contrario, el oponente, que se encontrará en desventaja, tratará de nivelar la partida, defendiéndose de los ataques rivales.

Los objetivos principales para los jugadores durante esta fase intermedia son:

- Definir una estrategia sólida de ataque/defensa en función de la situación en la que se encuentren.
- Desarrollar las piezas mayores, buscando especialmente mantener a la dama en posiciones que amenacen al rey rival.

- Consolidar la estructura de peones y defensa del rey creada durante la apertura.

Durante esta fase, cada pieza perdida por el jugador, supone un gran problema, especialmente si el jugador no había planeado la posibilidad de una captura por parte del rival. Por lo que, sopesar adecuadamente cada posible movimiento, se convierte en algo altamente necesario, es por esto, que es en esta fase en la que los jugadores más tiempo emplean en tomar cada decisión.

Cuando la estrategia de uno de los jugadores triunfa y comienza a amenazar las posiciones del rey, llegamos a la tercera fase de las partidas de ajedrez, el final del juego.

2.1.1.3 Final del juego

Esta es la fase que se corresponde con el desenlace de la partida. Como se ha comentado a esta fase, se llega tras haberse desarrollado las estrategias de ambos jugadores. La situación más habitual, llegado a este punto, es que sobre el tablero queden pocas piezas, un par de peones, algunas piezas menores, al menos una de las torres y eventualmente la dama.

En esta fase, una de las estrategias ha triunfado, por lo que el jugador con la estrategia adecuada, jugará para acorralar al rey y darle jaque maque. Por otro lado, el oponente se encargará de defenderse de los ataques.

Los principales objetivos para el atacante serán:

- Tratar de aislar al rey de sus piezas defensivas.
- Tratar de posicionar las mayores piezas propias cerca del rey, para así tener más posibilidades de ataque.
- Tratar de cortar las vías de escape del rey, para que poco a poco, este se vaya quedando sin escaques posibles a los que desplazarse, llegando finalmente al punto de jaque mate.

Los principales objetivos del defensor, será los opuestos a los del atacante:

- Tratar de mantener cerca del rey, el mayor número de piezas aliadas posibles, especialmente las torres, ya que se mueven con agilidad en un tablero despoblado de otras piezas.
- Tratar de bloquear las piezas de ataque enemigo para que así no pueda encerrar al rey.
- Tratar de abrir nuevas vías de escape al rey, atacando las piezas del oponente.

En el final del juego, también existe la posibilidad de que ambos jugadores lleguen con las fuerzas igualadas, y se conviertan a la vez en atacantes y defensores. Esta situación propicia, jugadas más “espectaculares”, ya que ambos jugadores deben considerar las jugadas valorando el compromiso de efectividad en el ataque frente a la exposición de esa pieza a ser capturada.

Una vez vistas las fases en las que se desarrolla una partida de ajedrez, se puede analizar cómo se producen los posibles movimientos de todas las piezas.

2.1.2 Movimientos posibles en ajedrez

No existe un consenso en el cálculo del número de posibles movimientos que se pueden dar en el juego del ajedrez. Sin embargo, se puede hacer un cálculo sencillo que permite ver de forma simple, y sin llegar a una cifra exacta, el gran número de jugadas disponibles que se pueden generar desde la posición inicial.

Inicialmente, los movimientos disponibles en el ajedrez son los movimientos de los peones 1 o 2 casillas. Hay 8 peones por lo que existen 16 movimientos disponibles de peones. Además, en los movimientos iniciales, se pueden mover los caballos, y cada caballo tiene dos posibles casillas a las que moverse, por tanto serían 4 movimientos más. En total, sobre el tablero inicial las blancas podrían hacer 20 movimientos. A esto le seguirían los 20 movimientos disponibles de las negras. Para el segundo turno de las blancas, los movimientos posibles serían los 20 iniciales, por todas las posibles combinaciones tras cada uno de los posibles movimientos iniciales.

De acuerdo con los cálculos de esta fuente, tras el segundo turno el número de posibles movimientos es 72.084, y tras el tercero habría más de 9.100.000 posibles movimientos. [Parra, 2013]

Nº Movimientos	Blancas	Negras
1	20	40
2	5.362	72.084
3	809.000	9.100.000

Tabla 1. Número de movimientos

Estos cálculos serían simplemente tratar de buscar el mejor movimiento a profundidad 3, desde la posición inicial. Sin embargo, durante una partida normal, estos datos serían mayores, debido a que durante el desarrollo de la partida, la posición inicial, en un turno concreto, desde la que se empieza a calcular no tiene sólo 20 movimientos disponibles, por lo que ya el número de movimientos iniciales sería mucho mayor.

Además a la hora de conocer todos los posibles movimientos de una partida de ajedrez, se debe entender, que existe una dificultad añadida, que hace que el cálculo de los movimientos totales que ofrece el ajedrez sea casi imposible. Esta dificultad se caracteriza porque a una posición dada, se puede llegar por más de una secuencia de movimientos, esto es lo que se conoce como transposiciones.

2.1.3 Transposiciones

Una transposición, es una secuencia de movimientos que alcanzan una posición determinada, que podría haber sido alcanzada por otra secuencia distinta. Por las reglas del ajedrez, y al igual que los movimientos, las transposiciones en ajedrez, son muy numerosas (una transposición es un movimiento, que se haya dentro del conjunto de movimientos totales).

Además, dentro de las transposiciones, podemos diferenciar dos tipos. Si la transposición se ha producido mediante una alteración en la secuencia de movimientos, se podría decir que se trata de una transposición de orden. Por el contrario, si la transposición se ha producido por una secuencia de movimientos

en el mismo orden pero pasando por distintas casillas, se podría decir que se trata de una transposición de escaque o de casilla.

Para verlo más claro, se toma como ejemplo la siguiente posición de ajedrez:

Ilustración 1. Posición ejemplo transposiciones

Esta es una situación irreal del tablero, sin embargo, permite ilustrar los distintos tipos de transposiciones. En este caso, se pueden dar los dos tipos de transposiciones previamente comentados:

- Transposición de orden: Ya que podríamos llegar a esta posición con más de una secuencia de movimientos. Por ejemplo, algunas transposiciones:
  - Peón e2 -> e4 / Peón g2 -> g4 / Alfil f1 ->e2 / Alfil e2 -> f3
  - Peón g2 -> g4 / Peón e2 -> e4 / Alfil f1 -> e2 / Alfil e2 -> f3
  - Alfil f1 -> e2 / Alfil e2 -> f3 / Peón g2 -> g4 / Peón e2 -> e4

Ilustración 4.Ejemplo 3 transposición de orden

- Transposición de casilla: Se podría llegar a la posición indicada inicialmente con, al menos, dos secuencias de movimientos:
    - Peón e2 -> e4 / Peón g2 -> g4 / Alfil f1 ->e2 / Alfil e2 -> f3
    - Peón e2 -> e4 / Peón g2 -> g4 / Alfil f1 ->g2 / Alfil g2 -> f3

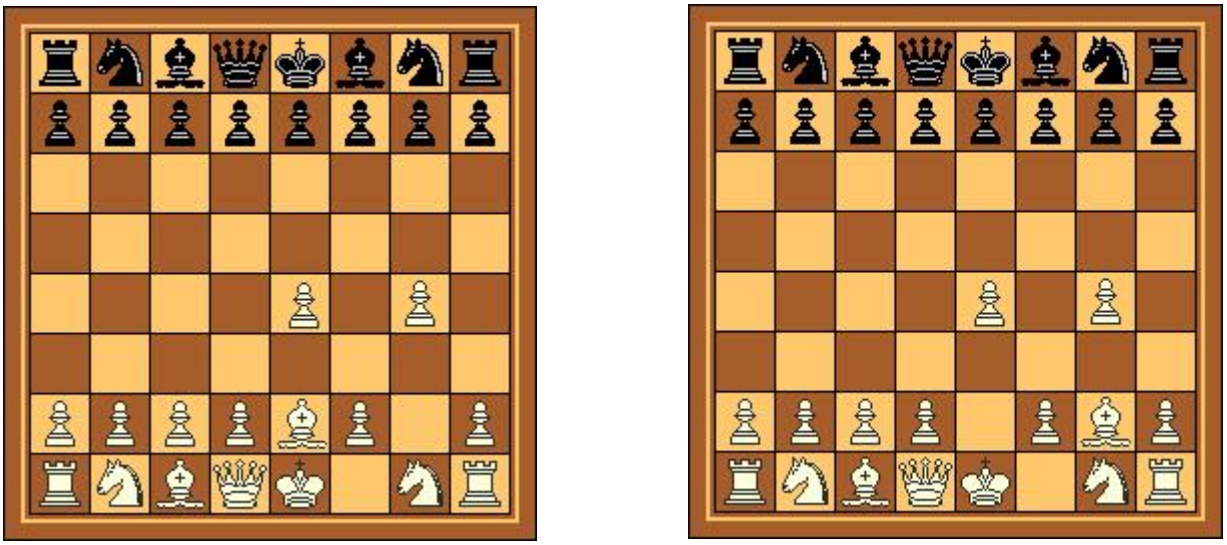


Ilustración 6. Ejemplo 2 Transposición de casilla

En ajedrez, las transposiciones, no son sólo importantes para el cálculo de posibles movimientos totales, sino que son de vital importancia en fases como la apertura, ya que, a una misma apertura, se puede llegar a través de muchas transposiciones de las anteriormente comentadas. Esto hace que tanto un jugador humano como un ordenador tengan ciertas dificultades para reconocer que una transposición se corresponde con una apertura conocida.

Una vez vistas todas las fases y todos los movimientos a tener en cuenta para el ajedrez, pasamos a ver como se clasifica a los jugadores de ajedrez.

2.1.4 Clasificación de los jugadores de ajedrez

Para saber cuáles son los mejores jugadores de ajedrez actualmente, es necesario ir a las clasificaciones publicadas por los distintos organismos internacionales relacionados con el juego del ajedrez.

También se debe comentar que para clasificar a los jugadores de ajedrez en función de su "calidad", se utiliza un sistema de puntación llamado ELO, en honor a su creador, Árpád Élő [Wikipedia, 2016]. Este es un método matemático, basado en un cálculo estadístico que indica la habilidad relativa de cada jugador. Este sistema de clasificación no sólo se usa en el ajedrez, sino que es utilizado en casi cualquier juego de estrategia, para clasificar a sus jugadores. Sin profundizar en los detalles exactos en los que se basa este sistema, los puntos ELO de cada jugador se modifican en función del resultado de la partida y del oponente contra el que juegue.

Existen varias federaciones que tienen su propia clasificación de puntos ELO, pero destacan FIDE [FIDE, 2015] (federación internacional de ajedrez) y USFC (federación de ajedrez de Estados Unidos). Dentro de la clasificación de puntos ELO, existen tres tipos de partida de ajedrez: clásica (90 minutos de tiempo total para cada jugador más treinta segundos por cada movimiento), rápida (60 minutos de tiempo total para cada jugador) y blitz (15 minutos de tiempo total para cada jugador).

La clasificación actual para partidas clásicas según FIDE es: [FIDE, 2015]

Standard Top 10 Players August 2016

Rank	Name	Title	Country	Rating	Games	B-Year
1	Carlsen, Magnus	g	NOR	2857	10	1990
2	Vachier-Lagrave, Maxime	g	FRA	2819	11	1990
3	Kramnik, Vladimir	g	RUS	2808	7	1975
4	Caruana, Fabiano	g	USA	2807	7	1992
5	Aronian, Levon	g	ARM	2792	0	1982
6	Nakamura, Hikaru	g	USA	2791	10	1987
7	So, Wesley	g	USA	2771	10	1993
8	Anand, Viswanathan	g	IND	2770	0	1969
9	Giri, Anish	g	NED	2769	10	1994
10	Karjakin, Sergey	g	RUS	2769	10	1990

Ilustración 7. Lista de los 10 mejores jugadores

También existe una clasificación de los motores de ajedrez, en función de los puntos ELO que estos han obtenido. Estos datos son importantes, porque la clasificación actual de los motores de ajedrez es esta: [CCRL, 2016]

Rank	Name	Rating			Score	Average Opponent	Draws	Games	LOS
		Elo	+	−					
1	Komodo 10.1 64-bit 4CPU	3375	+32	−31	78.6%	−190.4	40.2%	373	98.1%
2	Stockfish 7 64-bit 4CPU	3339	+16	−16	67.2%	−107.0	57.8%	1165	100.0%
3	Houdini 4 64-bit 4CPU	3255	+11	−11	55.9%	−38.4	50.9%	2633	99.7%
4	Andscacs 0.872 64-bit 4CPU	3211	+28	−28	49.6%	+4.7	56.5%	352	59.0%
5	Fire 4 64-bit 4CPU	3208	+13	−13	46.2%	+23.0	60.6%	1642	86.8%
6	Gull 3 64-bit 4CPU	3198	+12	−12	47.9%	+11.5	58.2%	2010	91.2%
7	Equinox 3.20 64-bit 4CPU	3185	+14	−14	43.5%	+41.6	60.7%	1429	97.1%
8	Critter 1.6a 64-bit 4CPU	3169	+9	−9	49.5%	+3.4	56.9%	4102	65.9%
9	Fritz 15 64-bit 4CPU	3165	+17	−17	42.4%	+48.5	59.8%	1038	70.5%
10	Rybka 4 64-bit 4CPU	3160	+11	−11	49.8%	−0.9	50.9%	2958	51.4%
11	Bouquet 1.8 64-bit 4CPU	3159	+12	−12	44.1%	+38.4	57.3%	2174	77.6%
12	Protector 1.9.0 64-bit 4CPU	3151	+18	−19	42.3%	+51.2	55.0%	880	52.3%
13	NirvanaChess 2.2 64-bit 4CPU	3150	+17	−18	40.9%	+59.4	57.7%	957	65.0%
14	Alfil 15.7 64-bit 4CPU	3145	+19	−19	40.4%	+64.6	55.0%	854	55.0%
15	Chiron 3.01 64-bit 4CPU	3143	+27	−27	47.9%	+11.6	62.0%	389	73.3%
16	BlackMamba 2.0 64-bit 4CPU	3134	+15	−15	45.1%	+29.7	55.2%	1311	62.5%
17	Fizbo 1.7 64-bit 4CPU	3129	+27	−27	56.1%	−38.9	52.5%	396	68.1%
18	Hannibal 1.5 64-bit 4CPU	3121	+19	−19	53.2%	−20.6	57.1%	819	80.0%
19	Strelka 5.5 64-bit	3111	+9	−9	52.6%	−13.9	55.5%	4558	67.5%
20	Naum 4.6 64-bit 4CPU	3107	+15	−15	47.5%	+15.3	54.0%	1410	

Ilustración 8. Lista mejores motores de ajedrez

Como podemos ver en la segunda clasificación, el mejor motor de ajedrez tiene 3375 puntos ELO, y el mejor jugador, actual campeón del mundo de ajedrez y record histórico de puntos ELO tiene 2857 puntos ELO, es decir, el mejor motor de ajedrez tiene 518 puntos ELO más que el mejor humano, con lo cual le ganaría con una alta probabilidad en todas las partidas en las cuales se enfrentasen. Incluso, el motor de ajedrez que se encuentra en el puesto 20 tiene 250 puntos ELO más que Magnus Carlsen (campeón del mundo de ajedrez). Estos datos nos permiten hacernos una idea de las capacidades que tienen los motores de ajedrez actuales.

2.1.5 Estructuras de datos para representar estados del ajedrez

Como se ha ido viendo durante la exposición de las fases de ajedrez y los movimientos disponibles durante el desarrollo del juego, si se quiere realizar un jugador de ajedrez en un ordenador, este tiene que manejar mucha información y gestionarla adecuadamente.

Para poder representar el estado del tablero de ajedrez en turno concreto existen numerosas alternativas, aunque se debe tener en cuenta que estas alternativas, deben ser realmente viables (no consumir un exceso de memoria o de tiempo). Además es necesario almacenar de alguna manera, los movimientos legales de cada pieza.

La primera opción de desarrollo, para representar el estado del tablero, sería la posibilidad de utilizar una estructura de memoria que resultase muy “visual”, es decir, cercana a lo que observa un jugador humano cuando ve un tablero de ajedrez. Esta estructura sería una estructura bidimensional, por ejemplo, un array de 8x8 posiciones, donde cada posición del array se corresponde con el estado de una casilla del tablero. Esta estructura tendría como ventaja la similitud con la forma de percibir el tablero como lo hacen los humanos, pero tendrían numerosos inconvenientes. Para empezar, es una estructura muy limitada que no permite guardar mucha información. Además, no es una estructura que esté optimizada para realizar ninguna de las operaciones que se necesitan más adelante, es decir, no sigue la forma de pensar de un ordenador.

Otra forma de contemplar el estado del tablero, similar a la anterior, pero algo más optimizada para un ordenador, sería la posibilidad de un único array lineal de 64 posiciones en la que se cada posición represente una casilla del tablero. Como principal ventaja presenta el almacenamiento de cada posición del tablero en una estructura relativamente sencilla de manejar para ordenador. Sin embargo, igual que en el caso anterior, no es lo suficientemente flexible como para cubrir toda la necesidad del sistema.

Otro punto común, a las dos estructuras que se acaban de describir, es decidir qué información se guarda en cada una de las posiciones del array. Hay distintas posibilidades:

- Se puede guardar únicamente si la casilla está o no ocupada. Aunque esta opción, gasta memoria para dar poca información.
- Se puede guardar el color de la pieza que está en esa casilla. Puede venir bien como referencia, aun así, ya que se gasta memoria en almacenar, se debe tratar de sacar el máximo provecho.
- Se puede guardar que pieza y de qué color, se encuentra en ese casilla. Esto se haría codificando cada pieza por ejemplo con un número concreto.

Otra posibilidad disponible para almacenar el estado del tablero, es almacenar una lista de nodos por cada pieza disponible en el tablero. Cada lista contendría todos los movimientos disponibles (nodos) de esa pieza, así cuando se quisiese hacer el movimiento de una pieza sólo habría que acudir a la lista de sus movimientos. Esta opción, ofrece la ventaja de tener en una misma estructura el estado del tablero y los movimientos disponibles. Sin embargo, es bastante costoso en tiempo, ya que para mantener actualizada esa lista por cada movimiento, se requieren muchas operaciones. Además, las listas habría que mantenerlas todas en memoria intermedia, lo cual supondría también cierto coste.

Como mejora a la estructura anteriormente comentada, es posible mantener una lista de casillas afectadas por un movimiento, así cuando se realice un movimiento, sólo habría que modificar aquellas listas, de aquellas piezas que se vean afectadas por ese movimiento. Eso incrementaría aún más el coste en memoria pero reduciría el coste de operaciones.

Otra características que tienen en común el uso de lista para almacenar el estado del tablero, es que ambas soluciones, necesitan una forma de calcular los movimientos disponibles para cada tablero particular, es decir, necesitan "acortar" su lista de movimientos legales, para obtener solo los movimientos disponibles en función del tablero existente. Por ejemplo, en un tablero con la disposición de piezas inicial, la torre tendría movimientos legales en global, pero no movimientos disponibles para esa situación del tablero en general.

Para solucionar este problema de "acortar" se puede tratar de sacar la situación de cada pieza sobre el tablero, a través de la lista de movimientos disponibles de cada pieza, aunque esto resultaría enormemente costoso en cuanto a las operaciones realizadas.

Otra solución que estaría disponible, sería la combinación de todas las nombradas anteriormente. Se podría utilizar un array general que mantuviese el estado actual del tablero, indicando en cada casilla el estado de esta (vacía o pieza y color), y utilizar además, una array lineal de 64 posiciones por cada tipo de pieza, y en cada posición del array una lista de todos los movimientos disponibles para esa pieza en esa casilla. De tal forma, que combinando ambas estructuras, lista de movimientos y array de tablero, se podría "acortar" rápidamente la lista de movimientos de una pieza para obtener sus movimientos disponibles. Además, esta combinación de array y lista tendría la posibilidad de que sólo se combinarían, cuando realmente se necesitase acceder a los movimientos disponibles de esa pieza, siendo esto muy eficiente en operaciones.

Utilizando los arrays de otra manera, se tendría una forma más de representar el tablero y los movimientos disponibles. Podría utilizarse un array o mapa lineal de 32 posiciones (una por pieza) en la que se indicase cada pieza en qué casilla se encuentra, y otro array o mapa, indicando si la casilla ha sido ya capturada o no. Combinando ambas estructuras podrían sacarse los movimientos disponibles para cada pieza. Esta solución es eficiente en memoria, ya que sólo consume dos arrays o mapas de 32 posiciones, sin embargo, requiere muchas operaciones para obtener todos los movimientos.

A continuación se muestra un esquema con las soluciones propuestas, y de acuerdo a lo comentado, se indica su nivel de consumo en memoria y tiempo, y la flexibilidad para utilizar esas estructuras.

	Memoria	Tiempo	Flexibilidad	Global
Array bidimensional	Baja	Alto	Baja	3/9
Array lineal	Baja	Alto	Baja	3/9
Lista movimientos	Medio	Medio	Medio	6/9
Array + lista	Medio	Bajo	Alta	8/9
Array de piezas + capturas	Baja	Medio	Baja	6/9

Tabla 2. Comparación de estructuras de memoria

Como se puede observar, según los criterios escogidos, la mejor opción en cuanto a la eficiencia a priori en tiempo y memoria y la flexibilidad ofrecida, la combinación de array y lista ofrece las mejores prestaciones.

2.2 Función de evaluación

Como ya se ha comentado, en este proyecto la estrategia para resolver el problema del ajedrez, es utilizar una función de evaluación que nos indique cómo de bueno es un movimiento. Así, evaluando todos los movimientos disponibles (al menos los más prometedores), esta función de evaluación, devolverá en cada posible movimiento un valor, y escogiendo de entre todos los movimientos disponibles el valor más alto, tendremos el mejor movimiento.

Actualmente, existen muchos motores de ajedrez que implementan funciones de evaluación capaces de batir incluso al actual campeón del mundo de ajedrez, como se ha visto anteriormente. De entre los tres primeros motores de ajedrez de la lista de mejores, sólo Stockfish es open source, los otros dos son comerciales.

De estos tres ejemplos de motores de ajedrez, cada uno desarrolla su propia estrategia para ganar la partida. Hay que destacar, que estos motores de ajedrez, están más orientados a ganar a otros motores de ajedrez que a ganar a humanos, ya que los humanos ya no suponen un reto para este tipo de software. Sin embargo, todos ellos son usados por los grandes jugadores de ajedrez para entrenarse y ver sus fallos. Cuando un motor de ajedrez se enfrenta a un

humano tiene distintos problemas que cuando se enfrenta a otra máquina. Analizar esta variedad de problemas también permite darse cuenta de cuáles son las limitaciones de la IA tal y como la conocemos ahora.

Estos tres motores de búsqueda, además de la función de evaluación ofrecen otras funcionalidades, como algoritmos de búsqueda propios o incluso interfaces. Sin embargo, como ya se ha dicho, lo realmente interesante para este proyecto es llamar a la función de búsqueda de este motor de ajedrez. Para hacer esto, existen diversos mecanismos.

Para conectar el motor de ajedrez con cualquier otro programa se podría utilizar la librería SWIG que ofrece C++, para hacer llamadas a C++ desde cualquier lenguaje de programación. Esta opción a priori resulta rápida, ya que únicamente sería necesario ejecutar el código de la función de evaluación, tiene un problema relativamente complejo, ya que para que la llamada a esa función de evaluación sea correcta, es necesario crear y rellenar todas las estructuras que utilice esta función de evaluación.

Otra posibilidad para conectarse a la función de evaluación, es utilizar la interfaz estándar UCI [WBEC, 2001].

2.1.1 UCI

UCI es una interfaz de comunicación entre un motor de ajedrez y cualquier otro programa que siga el estándar, incluyendo comunicación con interfaces gráficas, que fue desarrollado en 2000 por Rudolf Huber y Stefan Meyer-Kahlen.

Esta interfaz define una serie de normas, por las cuales programas escritos en otros lenguajes pueden conectarse a todas las funciones de evaluación [CPW, 2016]. De esta forma un programa escrito en un lenguaje diferente a C++, puede llamar al motor de ajedrez y ejecutarlo normalmente. Por tanto, si existe una interfaz que puede ejecutar todas las funcionalidades del motor de ajedrez, es posible utilizar esa interfaz para acceder a la función de evaluación entre otras cosas.

Esta interfaz define muchos parámetros para la comunicación, pero destacan dos de verdadera importancia. Depth, que indica la profundidad hasta la que el motor de ajedrez evalúa resultados, y por otro lado, MultiPV, indica las variantes de la rama de la mejor solución, para poder seguir buscando más soluciones.

Esta interfaz, evita además tener que rellenar expresamente, todas las estructuras de datos que necesita el motor de ajedrez [Mann, 2003]. Para ejecutar esta solución, únicamente es necesario llamar al ejecutable del motor de ajedrez, pasándole como argumento un string FEN [CPW, 2016].

2.1.2 Notación FEN

Este string es una notación estándar creada por David Forsyth, y que sigue el siguiente esquema.

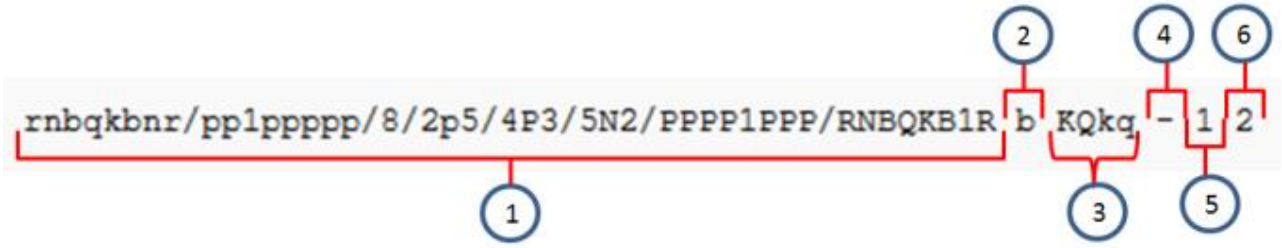


Ilustración 9. Explicación FEN

En la parte 1, se codifica todo el tablero según el siguiente esquema:

Torre Blanca	**R**	**Torre Negra**	**r**
Caballo Blanco	N	Caballo Negro	n
Alfil Blanco	B	Alfil Negro	b
Dama Blanca	Q	Dama Negra	q
Rey Blanco	K	Rey Negro	k
Peón Blanco	P	Peón Negro	p
Casillas vacías	Nº casillas seguidas	Separación de fila	/

Ilustración 10. Codificación FEN piezas

En la parte 2 se pone el jugador que le toca mover, ‘b’ si mueven blancas, ‘n’ si mueven negras.

En la parte 3 se especifican los posibles enroques, según el esquema:

No hay ningún enroque	-
Enroque Dama Blanca (enroque largo)	Q
Enroque Rey Blanco (enroque corto)	K
Enroque Dama Negra (enroque largo)	q
Enroque Rey Negro (enroque corto)	k

Ilustración 11. Codificación FEN piezas

En la parte 4, se especifican las capturas de peón al paso. Se escribe '-', si no hay posibilidad de captura de peón al paso, en caso de que hay posibilidad de captura, se pone la fila y la columna en minúsculas de la pieza que captura.

En la parte 5, se escribe el número de movimientos desde la última captura de pieza o movimiento de peón. Se entiende por movimiento, el cambio de posición de una pieza sea del color que sea, por tanto, un turno está formado por dos movimientos (uno de blancas y otro de negras). Este campo sirve para controlar la regla de los 50 movimientos.

En la parte 6, se escribe el número de turnos desde que se empezó la partida.

Una vez visto todos los mecanismos para conectar un motor de ajedrez con cualquier programa, pasamos a ver los motores mencionados anteriormente.

2.1.3 Komodo

Este motor de ajedrez con 3375 puntos ELO, es actualmente, el motor de ajedrez que encabeza la clasificación. Aunque esta clasificación es muy dinámica, podemos decir que es uno de los mejores motores de búsqueda actuales. [CPW, 2016]

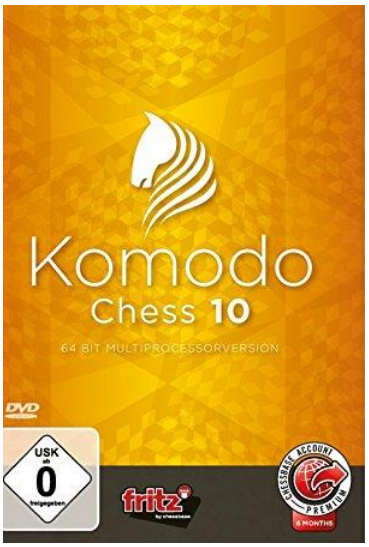


Ilustración 12. Komodo 10, motor de ajedrez

Fue desarrollado por Don Dailey y Larry Kaufman, ambos trabajaron en el MIT, y escribieron por separado distintos programas que jugaban al ajedrez, incluso Kaufman trabajaba en Rybka 3 (otro motor de búsqueda que llegó al ser el mejor del mundo hace unos años). Al juntarse, Dailey como programador y Kaufman como asesor de ajedrez crearon Komodo.

Debido a que Komodo es un software de pago, no se conoce con tanto detalle el código del programa ni la función de evaluación que utiliza. Sin embargo, si se

conocen alguna de las estrategias que utiliza. Como todos los motores de ajedrez, asigna puntos a cada pieza (torre, caballo, alfil,…) en función de si se encuentra en la apertura, el medio juego o en la fase final del juego.

Como explica Kaufman, Komodo tiene la capacidad de darle a cada pieza un valor más preciso que sus rivales llegando al medio juego con cierta ventaja derivada de esa correcta asignación de coeficientes [Hartmann, 2013].

Según las características vistas de Komodo podríamos destacar como fortalezas:

- Ha sido desarrollado por un gran maestro, por lo que los coeficientes de cada pieza están muy bien ajustados.
- Tiene un gran medio juego derivado de la característica anterior, y es capaz de lograr en esta fase una ventaja suficiente como para ganar seguro a un humano, y aventajar ligeramente a sus dos grandes competidores en el mundo de los motores de búsqueda.

Las debilidades de Komodo frente a los otros motores de búsqueda las podríamos resumir como:

- No saca ventaja en las aperturas debido a que en esta fase no influyen tanto los coeficientes de los jugadores, por lo que en partidas cortas que se desarrollen aperturas muy largas llega al medio juego con una ligera desventaja.
- En el final del juego no es capaz de hacer evaluaciones que permitan realizar tácticas tan profundas como Stockfish por lo que en partidas en las que el medio juego ha estado igualado, su estrategia en el final de la partida sería más débil y tendría problemas para batir a sus rivales.

2.1.4 Stockfish

Stockfish es un motor de ajedrez de código abierto escrito en C++ y desarrollado por Tord Romstad, Joona Kiiski y Marco Costalba. [Quora, 2015]

Ilustración 13. Stockfish 4, motor de ajedrez

Al igual que los otros motores de ajedrez analizados, Stockfish, asigna coeficientes a cada pieza para decidir su importancia. Además su función de evaluación, a la hora de escoger jugadas, siempre da mayor importancia a controlar las casillas centrales (defensa o ataque) frente a controlar las casillas desocupadas. También, da importancia a defender las posiciones de piezas propias en casillas centrales o atacar piezas enemigas cuando estas se hayan en casillas centrales.

Además es conocido, que asigna un bonus a las piezas en función de la casilla en la que se encuentren o las piezas que la rodeen.

Esto es algo que de una forma u otra todos los motores de ajedrez hacen y que se diferencia del razonamiento humano, ya que, tratar de controlar las casillas centrales durante una partida de ajedrez es algo básico que resulta muy evidente para cualquier jugador de ajedrez con cierta experiencia [CPW, 2016]. Sin embargo, entender que una pieza vale más por las piezas adyacentes y saber

el coeficiente de cuanto más vale, es algo realmente complejo, especialmente para un humano, que por el contrario un motor de ajedrez realiza con relativa facilidad y que le da una gran ventaja sobre un humano. Además, el bonus que le asigna cada motor de ajedrez a cada pieza es un parámetro que marca grandes diferencias a la hora de elegir un movimiento u otro.

Stockfish también tiene en cuenta las aperturas, donde prefiere desarrollar piezas menores y buscando el enroque tan pronto como sea posible. En este punto, los jugadores humanos suelen utilizar aperturas ya predeterminadas, llegando a conocer cientos de ellas y aplicándolas en función de los movimientos del rival, es decir, un motor de búsqueda tiene una heurística concreta para las aperturas pero no se basa en aperturas ya existentes, mientras que un jugador humano recuerda y se basa en aperturas ya existentes para decidir cuál es la mejor jugada. Aunque es cierto que en las aperturas el motor de búsqueda no saca una gran ventaja al humano, sí que le es útil para preparar el tablero para desarrollar durante el medio juego, la mejor estrategia posible.

Otra característica de Stockfish, es que no busca el resultado de tablas, por lo que trata siempre de llevar la iniciativa del juego, no defenderse. Esto provoca que tenga gran capacidad para desarrollar buenas estrategias al final del juego y ganar más partidas que sus rivales.

2.1.5 Houdini

Este motor de ajedrez con 3255 puntos ELO, es actualmente, el tercer motor de ajedrez de la clasificación. [Lagrain, 2012]

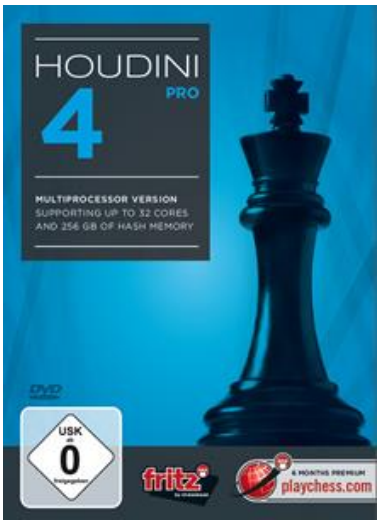


Ilustración 14. Houdini 4, motor de ajedrez

Houdini fue desarrollado por Robert Houdart y escrito en C++. El creador de Houdini es un informático y jugador de ajedrez que creó este motor de ajedrez inspirándose en los ya existentes para tratar de hacer algo diferente que fuese mejor que lo existente.

Houdini también es un software de pago pero esta algo más estudiado que Komodo debido a sus similitudes con otros motores de ajedrez de código abierto. Su creador, Robert Houdart, define Houdini como "un motor de ajedrez con tenacidad en posiciones difíciles y habilidad para defenderse en posiciones muy desfavorables hasta alcanzar el empate". De lo que deducimos que Houdini contempla, en ocasiones, jugar para quedar en tablas, siguiendo una estrategia conservadora. En este punto es totalmente distinto a Stockfish que no contempla la posibilidad de quedar en tablas. También se ha observado que

Houdini es capaz de jugar posiciones favorables adecuadamente siguiendo una estrategia de cortar todas las vías de escape al oponente.

Otra de las características observadas en Houdini en función a todas las partidas que ha disputado, es que tiene la capacidad de ver rápidamente las estrategias del rival y darse cuenta cual es el objetivo de este, tratando de anticiparse.

Según las características vistas de Houdini podríamos destacar como fortalezas:

- Capacidad de terminar en tablas partidas que el rival las calcula como ganadas. Cuando un motor de ajedrez calcula que tiene una amplia ventaja, es muy difícil que pierda esa partida, sin embargo Houdini es capaz de ganar esas partidas que el rival calcula como ganadas.
- Capacidad para ver y adaptarse rápidamente a las estrategias que está desarrollando el rival.
- Gran capacidad de juego al final de la partida en posiciones favorables.

Las debilidades de Komodo frente a los otros motores de búsqueda las podríamos resumir como:

- Poca capacidad de ataque. Houdini se puede clasificar como un motor de ajedrez conservador, que “prefiere” terminar en tablas a llevar el peso del ataque.
- Derivada de la característica anterior, Houdini suele sacar ventaja inicial al defenderse de los ataques, pero en el medio juego tiende a perder esa ventaja y llegar en desventaja al final de la partida, donde suele tratar de buscar tablas, lo que provoca que tenga menos victorias que sus dos principales competidores.

2.3 Algoritmos de búsqueda

Una de las principales limitaciones de la IA, que se ve muy claramente en el juego del ajedrez, es que la IA no entiende la realidad como nosotros. Cercano a cuestiones filosóficas, la pregunta sería qué es realmente la realidad o qué diferencia realmente un movimiento de otro. Los humanos no hemos sido capaces de resolver esta cuestión, por lo que hacérsela entender a un ordenador es prácticamente imposible. Así que, para un ordenador todos los movimientos posibles de un tablero son iguales, ya que no sabe diferenciar unos de otros. Con lo cual, ante un movimiento que parece evidente, debe realizar el mismo número de cálculos que ante uno no tan evidente. Llevándolo un poco más lejos, podría ser que un ordenador cayese en una trampa que un humano evitaría, ya que el humano razonaría simplemente que esa jugada es absurda.

Una de las herramientas utilizadas en IA, son los algoritmos de búsqueda. Esta estructura de memoria es utilizada para almacenar los resultados de las funciones de evaluación a profundidad superior a 1. La función de evaluación, supone una visión estática de la partida de ajedrez, es decir, es una cuantificación de un estado parcial. Sin embargo, con los algoritmos de búsqueda es posible ver y desarrollar las posibles estrategias disponibles para cada uno de los jugadores. Así, si se quiere dotar a un sistema de IA con la capacidad de realizar estrategias, es necesario, la implementación de un algoritmo de búsqueda.

Dentro de los algoritmos de búsqueda se pueden diferenciar varias clasificaciones. En este caso nos vamos a encargar de algoritmos de búsqueda

informada. Esto significa que al expandir los nodos vamos a saber el valor que nos va a dar la función de evaluación en cada nodo, no en todos los nodos, sino únicamente en los nodos hoja. En este apartado se va analizar los algoritmos planteados históricamente para resolver juegos de suma cero. Comenzando con el algoritmo del minimax que permite a un jugador maximizar sus resultados, luego se verá las mejoras que se pueden ir haciendo del algoritmo minimax y una implementación concreta de este, la poda alfa-beta. Además, se analizará los problemas derivados de aplicar un algoritmo como la poda alfa-beta, especialmente el problema del efecto horizonte, y tras esto, se detallarán las implementaciones propuestas a este problema del efecto horizonte como es la búsqueda quiescente.

2.2.1 Minimax

A la hora de resolver el problema de la partida de ajedrez, como se ha comentado anteriormente, que el ajedrez es un juego de suma cero. Por ello, es tan importante maximizar los puntos en los movimientos propios como minimizar los puntos en los movimientos del rival. Esto lleva a seguir una estrategia en la que suponiendo que el rival va a escoger siempre la jugada que maximice sus puntos, por tanto, la jugada peor posible para el jugador oponente. Estas observaciones están basadas en la teoría del minimax [Sion, 1958], que aplicadas al ajedrez las podemos resumir como, dentro de un árbol de búsqueda, se pretende escoger aquellos nodos de valor máximo en los

niveles que se correspondan con tu color y los nodos de valor mínimo en los niveles que se correspondan con el valor del oponente.

Esta teoría del minimax fue presentada inicialmente por John von Neumann en el año 1928 [Fan, 1952]. Aunque resulta algo evidente, ya que cuanto menos puntos consiga el oponente, más beneficio existe para el jugador, la formalización matemática no resulta tan trivial. Por ello, esta teoría de von Neumann fue una contribución realmente importante.

A partir de la teoría del minimax existen implementaciones en las que se combina con un árbol de búsqueda para obtener la poda alfa-beta que analizaremos más adelante.

2.2.2 Efecto horizonte Berliner

Otra cosa a tener en cuenta dentro de los algoritmos de búsqueda, es un problema que se genera cuando se acota la profundidad máxima de búsqueda en un algoritmo. Es el ya comentado "efecto horizonte" [CPW, 2016]. Este efecto fue descrito por Hans Berliner en 1973 [Berliner, 1973].

Este fenómeno se produce cuando el resultado devuelto por un algoritmo de búsqueda resulta condicionado por su profundidad máxima, es decir, se puede evaluar una jugada como buena o mala y no saber que a una profundidad mayor esta situación se revierte. Por ejemplo, puede ser que a profundidad 4 una jugada parezca la mejor, pero que a profundidad 6 esa jugada sea la peor porque el rival puede comer la reina (pieza más poderosa en el ajedrez moderno).

Otro problema derivado de este efecto horizonte [UIB, 2016], es que un algoritmo de búsqueda no es capaz de darse cuenta que hay situaciones en las que es mejor no elegir la jugada mejor para así evitar más adelante encontrarse en situaciones desfavorables. Por ejemplo, un algoritmo de búsqueda no es capaz de entender que hay movimientos en los que es mejor perder una pieza menor (peón) para salvar un par de movimientos más adelante una pieza mayor como puede ser la dama. Así, el algoritmo de búsqueda salvará la pieza menor y más adelante, según se desarrolle la partida terminará perdiendo la pieza mayor o incluso ambas.

Estos dos problemas son recurrentes en la IA y se dan debido a que es muy difícil que el ordenador entienda el problema que se le plantea más allá de las reglas predefinidas en la función de evaluación o el algoritmo de búsqueda. Si profundizamos un poco más, podemos darnos cuenta de que el verdadero problema radica en que el humano encargado de definir las reglas según las cuáles actúa la máquina, no es capaz de representar toda la compleja realidad del problema, con lo cual la máquina termina teniendo un comportamiento que en ciertas situaciones no se acerca al esperado en el espacio del problema.

Para solucionar el problema del efecto horizonte se utilizan otros tipos de algoritmo de búsqueda diferente de la poda alfa-beta, como pueden ser la búsqueda. Debido a que la poda alfa-beta es el algoritmo de búsqueda más usado en la resolución del problema del jugador de ajedrez y que la búsqueda quiescente es la principal solución al problema que plantea la poda alfa-beta, vamos a ver las ventajas y desventajas de cada uno, observando además en condiciones trabaja mejor cada uno.

2.2.3 Poda Alfa-Beta

El algoritmo de la poda alfa-beta se basa en el algoritmo del minimax. La mejora que hace este algoritmo, es eliminar aquellas partes del árbol que se sabe que van a ser irrelevantes, ya que nunca superarán al máximo local que se tenga en ese momento. Esta "poda" evita visitar un gran número de nodos, haciendo que la búsqueda sea más eficiente tanto en tiempo como en memoria [Fuller, 1973].

Inicialmente este algoritmo expande todos los nodos de todos los posibles movimientos de los que se disponga en el tablero hasta una profundidad determinada. Como ya se ha comentado, en el ajedrez las combinaciones son prácticamente infinitas, y expandir todos los posibles movimientos es altamente costoso tanto en memoria como en tiempo.

La solución a este problema, consiste en "podar" aquellas partes del árbol de búsqueda que ya se sabe que van a ser irrelevantes debido a que no van a mejorar al máximo local.

Este algoritmo utiliza dos variables alfa y beta.

- Alfa (α): Es el valor más alto de las opciones de maximización. Sólo se modificará alfa si se encuentra un valor más alto que el actual. Toma valor inicial -∞.

- Beta (β): Es el valor más bajo de las opciones de minimización. Sólo se modificará beta si se encuentra un valor más bajo que el actual. Toma valor inicial +∞.

Lo primero que hace el algoritmo es expandir en profundidad el primer nodo que se encuentre, hasta llegar a un nodo terminal o haber alcanzado la profundidad máxima, que en ese caso tomará el nodo actual como nodo terminal. Ahora, vemos que la profundidad máxima es 4. Y cuando llega al nodo terminal, ejecuta la función de evaluación sobre éste. Después ejecuta recursivamente hasta encontrarse otro nodo terminal. Como ambos nodos son terminales, ejecuta la función de evaluación sobre cada uno y sube el que tenga menor valor, así subiría el 5. Ahora mismo, el valor de beta es 5 y alfa -∞. [Wikipedia, 2016]

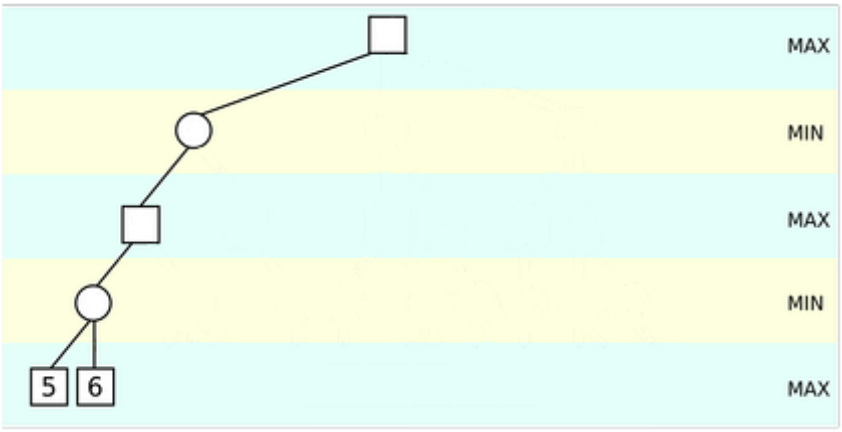


Ilustración 15. Ejemplo 1 poda alfa-beta

En la imagen, los nodos alfa (nodos de maximización, nodos cuadrados en azul) representan a la máquina calculando la mejor jugada, por tanto, tratando de maximizar su puntación. Por otro lado, los nodos beta (nodos de minimización, nodos circulares en amarillo) representan al jugador, del cual se trata de minimizar su puntuación.

En este punto, pasaría a la evaluación del siguiente nodo, pero siempre comparando el valor de los nodos terminales con el valor de beta (ya que nos encontramos en el nivel de minimización). En el momento que un valor de alfa sea menor o igual que beta se haría la poda, ya que ningún valor de los evaluados en esa parte del árbol podría modificar el valor de los nodos superiores, con lo que seguir evaluando los nodos de ese camino supondría malgastar tiempo y memoria.

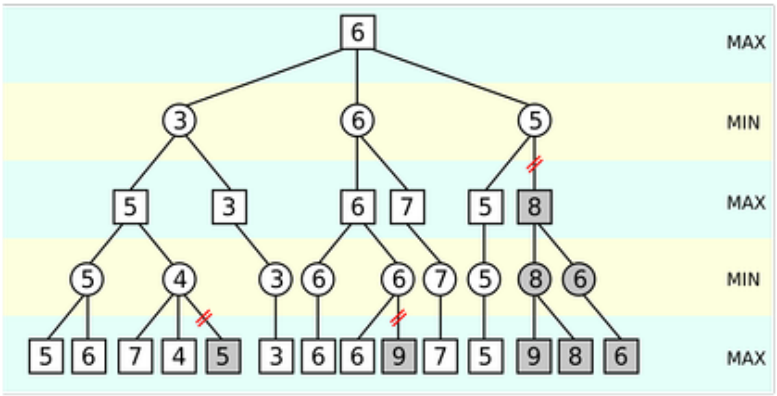


Ilustración 16. Ejemplo 2 poda alfa-beta

Como vemos en la imagen, uno de los sucesores del tercer nodo ha sido podado debido a que no iba a modificar el valor del mínimo local de ese subárbol. Al podar uno de los sucesores todo el subárbol de este resulta podado evitando expandir un gran número de nodos.

La mejora que supone la poda alfa-beta sobre el algoritmo del minimax, provoca que sea viable evaluar todos los movimientos "prometedores" a profundidades superiores a la evaluación que hace un humano. Sin embargo, aplicar la poda alfa-beta no siempre supone una ganancia constante, sino que el número de nodos puede variar según sea la distribución del árbol.

Lo peor que puede suceder con la poda alfa-beta, es que haya que evaluar un orden cuadrático de nodos, debido a que los valores de alfa y beta se vayan modificando de forma constante a lo largo del recorrido del árbol, es decir, que en ningún momento se produzca una poda. En este peor caso, tendríamos un número de nodos igual a b^n. Siendo b el número de movimientos iniciales, en caso de tener el tablero inicial, 40, y siendo n la profundidad a la que se quiere buscar. Por el contrario, si nos encontramos el mejor caso, el número de nodos a evaluar, sería por turno, el número de movimientos desde el tablero inicial (40) elevado al cociente superior de la mitad de la profundidad más el número de movimientos iniciales elevado a la función suelo de la mitad de la profundidad, y al resultado se le restaría uno.

En esta tabla podemos ver la diferencia que existe de valores en función de la profundidad a la que nos encontremos. Ya a partir de profundidad tres empezamos a ver la diferencia significativa de evolución que tienen ambos

casos, y descubrimos la importancia que tiene tratar de acercarnos lo máximo posible al mejor caso. [CPW, 2016]

number of leaves with depth n and b = 40		
depth	worst case	best case
n	b^n	$b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor} - 1$
0	1	1
1	40	40
2	1,600	79
3	64,000	1,639
4	2,560,000	3,199
5	102,400,000	65,569
6	4,096,000,000	127,999
7	163,840,000,000	2,623,999
8	6,553,600,000,000	5,119,999

Ilustración 17. Comparación de nodos poda alfa-beta

Una vez visto que acercarse al mejor caso puede suponer una diferencia de tiempo muy grande a la hora de realizar un movimiento, sería importante saber cómo diseñar el árbol de búsqueda para que expandiese el mínimo número de nodos posibles. Como podemos ver en el pseudocódigo, la poda se produce cuando:

```
si β≤α
        break (* poda αβ *)
```

Entonces debemos tratar de que desde el primer nodo beta sea menor que alfa, o dicho de otra manera, tratar de que alfa sea mayor que beta. Esta es una de las mejoras que se le pueden aplicar a la poda alfa-beta, conocida como ordenación dinámica, explicada a continuación.

2.2.3.1 Ordenación dinámica

Se trata de una implementación que mejora el rendimiento de la poda alfa-beta, observando que una buena ordenación de los nodos resulta verdaderamente beneficiosa en cuanto a tiempo. Esto se consigue ordenando el árbol de búsqueda de tal manera que los nodos con posibilidades de tener mayor puntuación se evalúen primero.

De tal forma, que con una ordenación adecuada se puede reducir la expansión de nodos varios ordenes de magnitud, y según aumente la profundidad más aumenta la reducción de tiempo. Para hacer esto, se hace una llamada a la función de evaluación en los nodos interiores y en función la respuesta se ordena de mayor a menor los nodos en los niveles MIN y de menor a mayor en los niveles MAX.

El objetivo principal de esta mejora es tratar de que el nivel superior escoja el primer valor de los ordenados, evitando así tener que evaluar el resto.

2.2.4 Búsqueda quiescente

Otra forma de afrontar el cálculo en profundidad de un movimiento, es la búsqueda quiescente [CPW, 2016]. Este algoritmo surgió para solucionar el problema ya ampliamente comentado del efecto horizonte. Así, se desarrolló una técnica que se acerca más a la forma en la que razona un humano a la hora de enfrentarse a un tablero de ajedrez. Sin embargo, este algoritmo también tiene algunos problemas durante su ejecución, ya que no resulta tan eficiente

como la poda alfa-beta, en los casos generales. La diferencia entre la búsqueda quiescente y la poda alfa-beta es que la poda alfa-beta evalúa todos los nodos que expanda de acuerdo a sus reglas y la búsqueda quiescente, por el contrario, sólo expande aquellos nodos que tenga actividad, entendiendo por actividad posibles capturas. Esto provoca que la búsqueda quiescente, no evalúe jugadas realmente buenas, y termine escogiendo un estado más "conservador" para así evitar perder una pieza, es decir, se podría decir que la búsqueda quiescente juega más a buscar las tablas. Además la poda alfa-beta para cuando encuentra un nodo hoja o cuando llega a su profundidad máxima, mientras que la búsqueda quiescente para cuando deja de haber actividad en un nodo, dicho de otra manera, cuando no se espera que en ese nodo haya capturas.

La búsqueda quiescente, como se ha comentado, sigue más el razonamiento humano, sin llegar elaborar estrategias, sí que diferencia los nodos en función de si necesita defenderse o atacar, o son nodos que simplemente no van a llevarle a ninguna captura. Cuando un humano está jugando al ajedrez descarta automáticamente movimientos que saben que o bien van a ser perjudiciales o que como mucho no le van a llevar a nada, y se dedica a expandir a gran profundidad aquellos nodos que intuye que le pueden permitir defenderse o atacar, en resumen, sólo expande aquellos nodos que le van a permitir seguir una estrategia que considera adecuada.

La búsqueda quiescente también se basa en el algoritmo del minimax al igual que la poda alfa-beta, ya que de los nodos que detecte con actividad va a necesitar decidir qué valor les va dando. Sin embargo, este algoritmo no expande en profundidad el primer nodo hasta que llega a un nodo terminal y

calcula su valor en la función de evaluación, sino que, en cada movimiento llama a la función de evaluación y mira cuales de los siguientes nodos a expandir provocan una captura de pieza tanto propia como ajena. También es necesario comentar, que al igual que en la poda alfa-beta, la ordenación del árbol de búsqueda condiciona el número de nodos que este expanda, ya que si ordenamos el árbol de tal forma que los primeros nodos que se expandan contengan las mejores capturas, podremos empezar a "podar" cuanto antes el árbol y así evitar las capturas de piezas menos prometedoras.

En este diagrama podemos ver una comparación de cómo funciona un árbol de búsqueda normal que expandiría todos los nodos en amplitud. Este árbol de la figura también podría ser un árbol de una poda alfa-beta en caso de que haya tenido que expandir todos los nodos. En los niveles que están a mayor profundidad podemos ver cómo trabaja la búsqueda quiescente que se limita a expandir nodos "con actividad". [CPW, 2016]

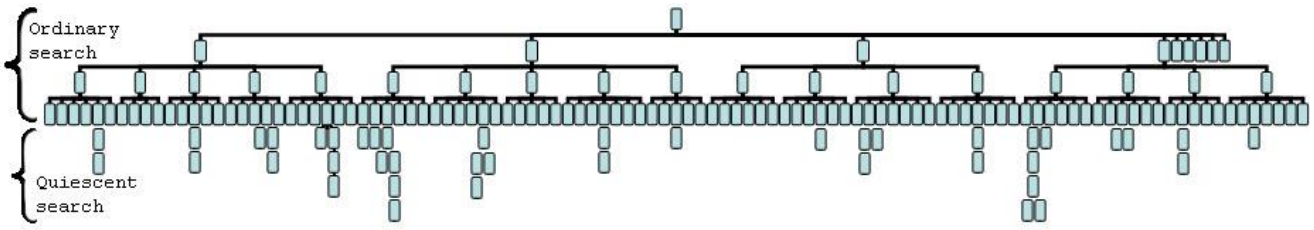


Ilustración 18. Poda alfa-beta vs Búsqueda quiescente

Otro aspecto a tener en cuenta, es que hay algunos motores de ajedrez que utilizan de forma combinada ambos algoritmos. Cuando la poda alfa-beta termina de evaluar todos los nodos terminales, se aplica la búsqueda quiescente para evitar el efecto horizonte y evitarse así la pérdida de piezas no detectada durante la poda alfa-beta.

Capítulo 3.

Objetivos

El objetivo principal de este proyecto es el desarrollo y análisis de un jugador de ajedrez. El jugador tendrá la capacidad de jugar de forma autónoma utilizando una biblioteca de aperturas, una función de evaluación y eligiendo la mejor jugada con el uso de un algoritmo de búsqueda. Además se analizarán los resultados obtenidos por el algoritmo jugando contra el motor de ajedrez Stockfish a distintas profundidades.

A continuación se detalla la lista de objetivos:

- **Software en Go:** Desarrollar en el lenguaje de programación Go, un software capaz de jugar al ajedrez siguiendo las reglas de la Federación Internacional de Ajedrez.
- **Función de evaluación:** Conectar el software anteriormente mencionado a un motor de ajedrez llamado Stockfish para utilizar su función de evaluación, y así saber el valor de cada movimiento disponible
- **Algoritmo de búsqueda:** Utilizar un algoritmo de búsqueda: poda alfa-beta, para encontrar la mejor solución a profundidad mayor que 1.
- **Análisis algoritmos:** Analizar los resultados obtenidos por el algoritmo de búsqueda y la biblioteca de aperturas enfrentándose al motor de ajedrez Stockfish, a varias profundidades.
- **Aperturas:** Implementar un sistema capaz de recibir una lista de aperturas y decidir entre ellas cuales deben ser descartadas y cuál es la mejor para ser ejecutada como siguiente movimiento.
- **Conceptos:** Comprender el funcionamiento de los algoritmos de búsqueda aplicados a un problema de IA, para que luego puedan ser extrapolados a cualquier otro ámbito o problema.

Capítulo 4.

Desarrollo

En este apartado se va a describir el proceso seguido para analizar el problema, las decisiones tomadas sobre este análisis, y cómo se ha construido el sistema que soluciona el problema. El proceso se divide en:

- **Análisis del problema:** En esta sección vemos cuáles son las partes en las que se puede dividir el problema y así estudiarlo de forma más sencilla. Además se definen los requisitos y casos de uso que se dan en cada parte del problema.

- **Decisiones tomadas:** En esta sección se detallan y justifican las decisiones tomadas según el análisis que se ha hecho del problema.

- **Diseño e implementación del sistema:** En esta sección se explican todos los componentes que conforman el sistema que se ha diseñado para solucionar el problema inicial.

4.1 Análisis del problema

El problema que se plantea en este proyecto se puede dividir en distintas partes para poder encontrar la solución más adecuada en cada uno de los subproblemas. Durante la realización del proyecto se pretende crear y analizar un jugador de ajedrez.

Para crear un jugador de ajedrez primero se debe crear un programa capaz de jugar de forma autónoma a este juego. Para ello, necesitamos un software que

tenga programadas todas las reglas que existen en el ajedrez moderno, es decir, que sea capaz de conocer todos los movimientos disponibles en cada jugada tanto del jugador como del oponente.

La implementación de este jugador no es una tarea trivial, ya que se necesita crear una estructuras de memoria que sean capaces de almacenar en todo momento las posiciones del tablero, y sobre estas posiciones, sacar cuáles son los movimientos disponibles de cada pieza o saber que movimientos aun estando disponibles son ilegales, por ejemplo, un peón puede tener un movimiento disponible de avanzar pero este ser ilegal porque el rey se encuentra en situación de jaque.

Además hay que diseñar el sistema de tal forma que las estructuras sean más eficientes en tiempo que en memoria. En el problema que se trata de resolver, la variable crítica es el tiempo, ya que se necesita que el sistema ejecute movimientos a una alta profundidad en el menor tiempo posible (cuanto menos tiempo se use en ejecutar un buen movimiento, menos tiempo tiene el oponente para decidir su jugada). En general en los ordenadores modernos, la memoria no suele ser un problema, por lo que tener un gran número de estructuras almacenadas en memoria intermedia no debería suponer un desafío para un ordenador actual. Por ello, una buena decisión sería precargar todas las estructuras posibles en memoria intermedia para que a la hora de ejecutar el programa únicamente se ocupe leer estas estructuras y llamar a otros módulos del sistema.

Como ya se ha explicado, para ayudar al programa a jugar se van a implementar una serie de estructuras que permitan al programa hacer movimientos de acuerdo a aperturas ya conocidas, por lo que habrá que diseñar cómo se va a realizar esta funcionalidad.

Una de las complicaciones que tiene este módulo es encontrar una fuente de aperturas lo suficientemente grande como para que contemple la gran mayoría de las aperturas conocidas con sus respectivas variantes. Otra de las complicaciones, derivada de la anterior, es que si la fuente de aperturas no fuese lo suficientemente grande, o incluso aunque fuese muy extensa, podría darse que hubiese permutaciones de aperturas no contempladas. Esta situación provoca que el programa desperdicie la oportunidad de jugar una apertura y comience a utilizar las heurísticas lo que ralentiza considerablemente su tiempo de decisión de movimiento.

Una vez que el programa es capaz de jugar y realizar unas aperturas, se le debe dotar de la IA que le permitirá jugar de forma autónoma. En este punto, entra en juego el motor de ajedrez con su función de evaluación, que indicará al programa cuál es el mejor movimiento de entre los disponibles en cada caso.

En cuanto a la función de evaluación utilizada, se debe tener en cuenta que sea una función con un número de puntos ELO elevado, esto permitirá que el valor asignado sea mejor y por tanto los movimientos más adecuados. Además una de las características observadas al analizar los motores de ajedrez, es que los tres mejores están escritos en el lenguaje C++, por lo que si el programa estuviese desarrollado en un lenguaje diferente, habría que conectar de alguna manera la llamada a la función de evaluación con el programa en sí. También, como

característica deseable tendríamos que el motor de ajedrez escogido sea gratuito y open source, para poder realizar una implementación que pueda ser mantenida por personas distintas al desarrollador, lo cual iría en consonancia con el resto del proyecto, en el que todos los recursos utilizados son gratuitos.

Cuando el programa es capaz de asignarle un valor a cada movimiento que le indica cómo de bueno es este, necesitamos que el programa no sólo elija el mejor movimiento valorando únicamente los disponibles en ese momento, sino que necesitamos que el programa simule una estrategia. Esto se consigue dándole la capacidad al programa de evaluar las soluciones en profundidad, es decir, que vea que resultados se obtienen ejecutando en cada caso una secuencia de movimientos.

En esta parte del proyecto, debemos tener en cuenta el parámetro crítico de la ejecución que es el tiempo. Un árbol de búsqueda va a expandir muchos nodos para encontrar la mejor solución, y cuantos más nodos expanda a mayor profundidad más acertada será esta, por ello, más posibilidades de tomar ventaja sobre el oponente. Pero, también hay que tener en cuenta, que a mayor número de nodos y mayor profundidad, mayor tiempo de ejecución. Por tanto, se debe encontrar un equilibrio entre el tiempo que emplee el programa en decidir qué movimiento ejecutar y la profundidad a la que busca este movimiento.

La última subdivisión que podemos hacer de este proyecto, es el apartado de pruebas. Ya se ha comentado que uno de los objetivos es probar un algoritmo de búsqueda que ejecute a cierta profundidad y con ayuda de las aperturas, frente a un motor de ajedrez con alta puntuación ELO. Además las pruebas deben implementarse de tal forma que testeen todas las funcionalidades del programa con el objetivo de encontrar errores. Por ello, deben diseñarse pruebas con dos objetivos, unas con el objetivo de comprobar que el sistema no tiene fallos, y unas segundas pruebas, las cuales deben tener como objetivo probar el sistema en un entorno más realista y que permitan observar diferencias entre los algoritmos de búsqueda y ver a qué situaciones se adapta mejor cada uno de ellos.

4.1.1 Casos de uso

Los casos de uso nos ayudan a entender y ejemplificar mejor las posibles funcionalidades del sistema en la interacción el usuario.

En este caso hay dos actores:

- Jugador: jugador humano de ajedrez que se comunique con la máquina, si bien este jugador puede usar el programa tanto para jugar contra él, como para probar los algoritmos de búsquedas con distintos tableros, sin embargo, sea cuál sea la funcionalidad que pruebe es un mismo usuario, con características idénticas.
- Motor de ajedrez: Este actor es el motor de ajedrez del que se utiliza su función de evaluación.

El diagrama de los casos de uso es el siguiente:

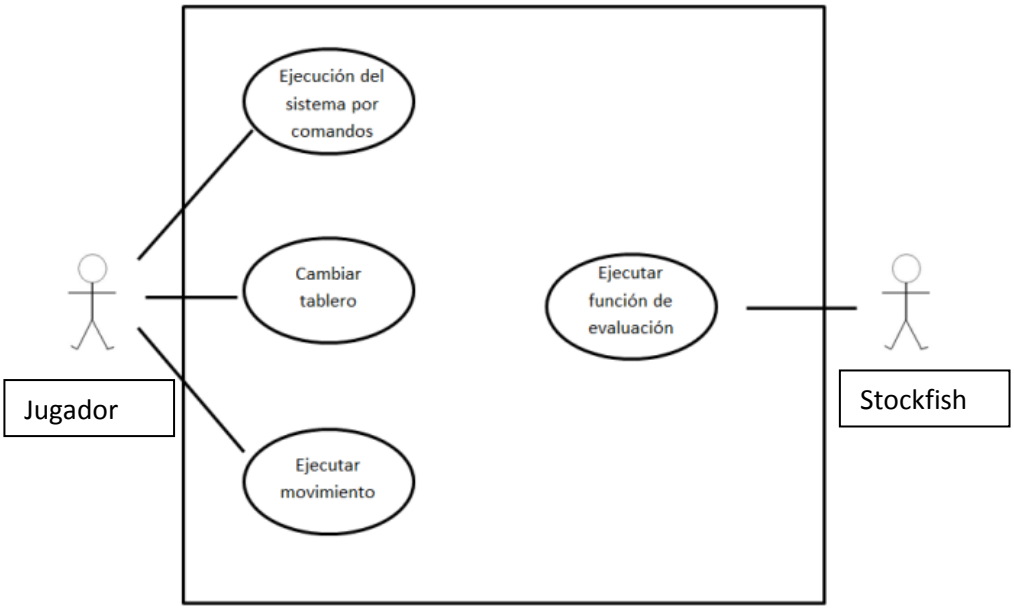


Ilustración 19. Casos de uso

El esquema que se va a seguir para los casos de uso es:

- **Identificador**: Este campo catalogará los casos de uso asignándoles un código. Este código estará compuesto por dos campos siguiendo esta estructura:

  CU-NNN **NNN** – será el número del caso de uso

- **Nombre del caso de uso:** Este nombre permitirá hacerse una idea del caso de uso de forma rápida.
- **Actor:** Actores involucrados en este caso de uso.

- **Descripción del caso de uso:** En este campo se hará una descripción del caso de uso que permita comprender su existencia y su necesidad.
- **Precondiciones:** Estas son las condiciones que deben darse para poder ejecutar el caso de uso
- **Secuencia:** Este apartado describe la lista de acciones a completar para llevar a cabo el caso de uso.

A continuación se muestra una tabla de ejemplo, para ver más claramente cómo se van a componer las tablas de los casos de uso.

CU-000	Tabla ejemplo caso de uso
Actor	Actor involucrado en el caso de uso
Precondiciones	Precondiciones que se dan en el caso de uso
Descripción	Descripción del caso de uso
Secuencia	Secuencia de ejecución en el caso de uso

Tabla 3. Ejemplo tabla caso de uso

CU-001	Ejecución
Actor	Jugador
Precondiciones	Ejecución por comandos del código
Descripción	Este caso de uso es el paso inicial para ejecutar el jugador de ajedrez a través de la línea de comandos
Secuencia	1. Compilar el programa en caso de no estar ya compilado 2. Definir los argumentos de entrada necesarios para el programa 3. Ejecutar el programa y esperar los resultados de salida

Tabla 4. CU-001

CU-002	Cambiar tablero
Actor	Jugador
Precondiciones	Ejecución por comandos del código
Descripción	Este caso permite pasarle como tablero inicial, cualquier tablero que el jugador desee, incluso tablero con movimientos ya realizados
Secuencia	1. Editar el fichero de texto en el que el programa recibe el tablero 2. Ejecución por comandos con los parámetros escogidos

Tabla 5. CU-002

CU-003	Ejecutar movimiento
Actor	Jugador
Precondiciones	Recibir movimiento de salida del programa
Descripción	Este caso de uso permite al usuario ejecutar el movimiento deseado
Secuencia	1. Se espera a que el programa decida que jugada ejecutar 2. Una vez leída la jugada ejecutada por el programa el usuario decide que jugada ejecutar. 3. Escribe en la consola la jugada siguiendo pieza, posición inicial y posición final.

Tabla 6. CU-003

CU-004	Ejecutar función de evaluación
Actor	Stockfish
Precondiciones	Tener un estado ya preparado para pasarle al motor de ajedrez
Descripción	Este caso de uso permite conocer la evaluación del motor de búsqueda en un estado parcial pasado como parámetro
Secuencia	1. Se genera el String FEN para pasárselo a Stockfish. 2. Se crea las estructuras necesarias para ejecutar Stockfish. 3. Se ejecuta Stockfish con el string FEN como argumento. 4. Se recibe y procesa los datos devueltos por Stockfish.

Tabla 7. CU-004

4.1.2 Requisitos del sistema

Una vez analizados todas las partes del problema pasamos a definir los requisitos, que se van a clasificar en tres categorías:

- **Requisitos de sistema:** son los requisitos del programa encargado de mantener las estructuras de memoria del tablero y las piezas, así como comprobar que todos los movimientos son correctos.

- **Requisitos de agente:** son los requisitos del sistema de inteligencia artificial encargado del comportamiento autónomo del programa.

- **Requisitos de pruebas:** son los requisitos de las pruebas que se van a realizar para comprobar que el sistema está totalmente correcto.

En cada una de las categorías de requisitos anteriores, existe a su vez una división entre requisitos funcionales (relacionados con la funcionalidad del sistema) y no funcionales (relacionados con las restricciones del sistema).

Cada requisito estará representado en una tabla que seguirá un esquema como el que se muestra a continuación.

- **Identificador**: Este campo catalogará los requisitos asignándoles un código. Este código estará compuesto por tres campos siguiendo esta estructura:

XXXX – tipo de requisito (PROG es requisito de programa, AGEN es requisito de agente y PRUE es requisito de pruebas)

XXXX-L-NNN **L** – requisito funcional (F) o no funcional (NF)

NNN – será el número de requisito de esa categoría

- **Nombre del requisito:** Este nombre permitirá hacerse una idea del requisito de forma rápida.
- **Descripción del requisito:** En este campo se hará una descripción del requisito que permita comprender su existencia y su necesidad.

Un ejemplo de cómo quedaría un requisito:

PROG-F-000	Requisito de ejemplo
Descripción	Este es un requisito de ejemplo para comprender la estructura de los requisitos

Tabla 8. Ejemplo tabla requisito

4.1.2.1 Requisitos de programa

PROG-F-001	Lectura tablero
Descripción	El sistema será capaz de leer, de un fichero de texto, el tablero considerado como inicial. La ruta del fichero se pasará como parámetro al ejecutable.

Tabla 9. PROG-F-001

PROG-F-002	Elección algoritmo de búsqueda y profundidad
Descripción	El sistema será capaz de leer, como parámetro del ejecutable, la profundidad máxima del algoritmo de búsqueda.

Tabla 10. PROG-F-002

PROG-F-003	Lectura de aperturas
Descripción	El sistema será capaz de leer, de un fichero de texto, la lista completa de aperturas.

Tabla 11. PROG-F-003

PROG-F-004	Estructuras de datos
Descripción	El sistema será capaz de crear y gestionar una estructura de datos por cada pieza del tablero, donde se almacenará el estado de la pieza.

Tabla 12. PROG-F-004

PROG-F-005	Estado del tablero
Descripción	El sistema será capaz de almacenar en una o varias estructuras de datos el estado actual del tablero.

Tabla 13. PROG-F-005

PROG-F-006	Movimientos ilegales
Descripción	El sistema será capaz de detectar movimientos ilegales a través de la estructuras de datos que almacenen el estado de cada pieza y del tablero.

Tabla 14. PROG-F-006

PROG-F-007	Movimientos disponibles
Descripción	El sistema será capaz de generar una lista con todos los movimientos disponibles.

Tabla 15. PROG-F-007

PROG-F-008	Movimientos de aperturas
Descripción	El sistema será capaz de seguir una secuencia de movimientos de una apertura que sea ejecutada por él o por el oponente.

Tabla 16. PROG-F-008

PROG-F-009	Función de evaluación
Descripción	El sistema será capaz de conectarse a un motor de ajedrez para obtener su función de evaluación y devolvérselo al propio sistema, a través del UCI.

Tabla 17. PROG-F-009

PROG-F-010	Movimientos realizados
Descripción	El sistema será capaz de mostrar por pantalla el movimiento recientemente realizado por el sistema.

Tabla 18. PROG-F-010

PROG-F-011	Movimientos a realizar
Descripción	El sistema será capaz de recibir por teclado el siguiente movimiento a realizar por el sistema.

Tabla 19. PROG-F-011

PROG-NF-001	Estructuras eficientes
Descripción	El sistema implementará estructuras eficientes en tiempo según los requisitos definidos.

Tabla 20. PROG-NF-001

PROG-NF-002	Requisitos Hardware
Descripción	El programa será capaz de ejecutar sobre un sistema que tenga como mínimo 4 GB de RAM y como CPU i5 o similares.

Tabla 21. PROG-NF-002

4.1.2.2 Requisitos de agente

AGEN-F-001	Valor retorno
Descripción	El sistema será capaz de devolver un valor generado por la función de evaluación para cada movimiento pasado como parámetro.

Tabla 22. AGEN-F-001

AGEN-F-002	Estadísticas
Descripción	El sistema será capaz de devolver las estadísticas de tiempo y nodos expandidos durante la búsqueda del mejor movimiento.

Tabla 23. AGEN-F-002

AGEN-F-003	Profundidad de búsqueda
Descripción	El algoritmo de búsqueda será capaz de buscar como mínimo a profundidad 4 en el árbol de búsqueda.

Tabla 24. AGEN-F-003

AGEN-F-004	Parámetros de profundidad
Descripción	El algoritmo de búsqueda será capaz de adaptarse a los parámetros de profundidad especificados como parámetros.

Tabla 25. AGEN-F-004

AGEN-F-005	Resolución de movimientos
Descripción	El sistema será capaz de resolver cualquier tablero especificado como parámetro en la profundidad marcada.

Tabla 26. AGEN-F-005

4.1.2.3 Requisitos de prueba

PRUE-F-001	Resultado de pruebas por nodos
Descripción	El sistema deberá indicar cuál de los dos jugadores (blancas o negras) expande menos nodos.

Tabla 27. PRUE-F-001

PRUE-F-002	Resultado de pruebas por tiempo
Descripción	El sistema deberá indicar cuál de los dos jugadores (blancas o negras) emplea menos tiempo.

Tabla 28. PRUE-F-002

PRUE-F-003	Resultado de pruebas por solución
Descripción	El sistema deberá indicar cuál de los dos jugadores (blancas o negras) gana la partida.

Tabla 29. PRUE-F-003

PRUE-F-004	Cobertura de las pruebas
Descripción	Las pruebas deberán cubrir todos los posibles movimientos de todas las piezas existentes.

Tabla 30. PRUE-F-004

PRUE-NF-001	Equipo de pruebas
Descripción	Se harán cierto número de pruebas sobre equipos que no cumplan el estándar mínimo de RAM y CPU para observar el resultado.

Tabla 31. PRUE-NF-001

PRUE-NF-002	Software de pruebas
Descripción	Las pruebas se harán sobre distintas versiones del lenguaje de programación, para comprobar la capacidad de actualización del sistema.

Tabla 32. PRUE-NF-002

4.2 Decisiones tomadas

En este apartado se detallan las decisiones tomadas, en base al estudio del estado de las tecnologías actuales y en base a los requisitos definidos sobre el sistema que se va a desarrollar para tratar de solucionar el problema inicial. Las decisiones tomadas se pueden enmarcar en cuatro grandes categorías: lenguaje, aperturas, función de evaluación y algoritmo de búsqueda.

4.2.1 Lenguaje

En este apartado se detallan las decisiones tomadas sobre el lenguaje en el que se va a escribir el jugador de ajedrez, explicando por qué han sido descartados otros y elegido éste.

El lenguaje elegido para desarrollar este proyecto es Go [Griesemer, 2009], también conocido como Golang. Es un lenguaje desarrollado por Google en 2007 y cuyos creadores son Robert Griesemer, Rob Pike, and Ken Thompson [Griesemer, 2009]. Este lenguaje está basado en otros lenguajes anteriores bastante famosos como C/C++, Pascal/Modula/Oberon.

Es un lenguaje compilado, no orientado a objetos (aunque puede simularse con bastante precisión la orientación a objetos), con un “garbage collector” y posibilidad de computación paralela [Pike, 2012]. Como se puede apreciar tiene todas las características de los lenguajes compilados de java, sin embargo, tiene como objetivo tener la sencillez y ligereza de los lenguajes interpretados como javascript o phyton tal y como explica Google en el blog dedicado al lenguaje de programación.

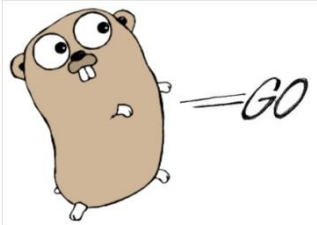


Ilustración 20. Icono del lenguaje Go

Go dispone de las estructuras de programación básicas y comunes a los lenguajes en los que está basado. Dispone de array, arrays redimensionables (slices), listas, mapas, etc. Como se comenta en la documentación Go, todavía no dispone de un amplio abanico de estructuras de memoria o de tipos de genéricos, porque los desarrolladores del proyecto, entienden que la implementación de estas estructuras incluiría mucha complejidad, sin aportarle grandes cambios al lenguaje. Aun así, ellos entienden que según vaya creciendo el lenguaje, y este vaya cubriendo más necesidades, se hará necesario incluir más estructuras.

Otra de las razones por la que se ha escogido este lenguaje para desarrollar este proyecto es que Go es un lenguaje totalmente open source, lo cual permite que junto con los desarrollos que va haciendo Google del lenguaje, exista una inmensa comunidad de desarrolladores que apoyen este proyecto. Lo que hace que Go tenga unas grandes expectativas de crecimiento y desarrollo en el futuro, además de estabilidad y durabilidad. Todos estos aspectos son de vital importancia, ya que cuando un desarrollador diseña su código, uno de los aspectos que tiene en cuenta es que este pueda ser mantenido en el tiempo, es decir, que la vida del proyecto sea lo más larga posible, y esto es posible debido

a la mucha gente implicada en el proyecto (no sólo profesionales de la empresa Google).

Además de la opción de desarrollar en Go, se tuvieron en cuenta otras opciones como Java o C++.

Java fue descartado principalmente por problemas de rendimiento, debido a que el manejo que hace internamente de la memoria ralentiza el programa en general. Es ampliamente conocido que una de las grandes ventajas de java es su capacidad de ser multiplataforma, pudiendo ejecutar el mismo código independientemente de la máquina en la que se encuentre. Sin embargo, esto que resulta una ventaja en cuanto a la portabilidad se torna en desventaja en cuanto al rendimiento, llegando a tener diferencias significativas en programas que utilizan mucha memoria. En este proyecto concretamente, el uso de memoria es significativo, tanto a la hora de almacenar el estado de cada pieza y el tablero, como a la hora de ejecutar los árboles de búsqueda en los que se expanden muchos nodos con un considerable gasto de memoria. En esta comparación podemos ver las diferencias que existen en la ejecución entre ambos. [Benchmarks, 2014]

Ilustración 21. Logo Go vs Logo Java

mandelbrot

source	secs	KB	gz	cpu	cpu load
Go	**5.68**	30,756	894	22.56	100% 100% 99% 99%
Java	7.14	88,236	796	27.93	97% 98% 98% 99%

Ilustración 22. Go vs Java. Mandelbrot

pidigits

source	secs	KB	gz	cpu	cpu load
Go	**2.80**	11,352	685	2.80	47% 9% 1% 45%
Java	3.11	33,364	938	3.20	99% 2% 2% 2%

Ilustración 23. Go vs Java. Dígitos de Pi

En estas dos tablas, podemos ver una comparación entre el tiempo, los KB de RAM y la carga de la cpu que utiliza cada uno de los lenguajes en la resolución de tres problemas distintos. El primero es la generación del conjunto fractal de mandelbrot y el segundo es el cálculo de los dígitos de Pi. En ambos casos vemos como el tiempo de ejecución de Go es inferior.

Por otra parte, C++ es un lenguaje muy parecido a Go, ampliamente utilizado. Además la función de evaluación escogida está escrita en C++, por lo que conectar ambos módulos del proyecto sería sencillo. Sin embargo, Go ofrece una mayor simplicidad a la hora de implementar los mecanismos para hacer funcionar el sistema.

Además si comparamos el rendimiento de ambos lenguajes en tiempo de ejecución de problemas similares a los planteados en este proyecto, veremos que aunque C++ es superior no es tan claramente superior a Go, por lo que la eficiencia que se pierde en el tiempo de ejecución, se compensa con las ventajas ya comentadas de Go.

Ilustración 24. Logo Go vs Logo C++

mandelbrot

source	secs	KB	gz	cpu	cpu load
Go	**5.68**	30,756	894	22.56	100% 100% 99% 99%
C++ g++	5.82	33,952	726	22.40	96% 95% 95% 100%

Ilustración 25. Go vs C++. Mandelbrot

pidigits

source	secs	KB	gz	cpu	cpu load
Go	2.80	11,352	685	2.80	47% 9% 1% 45%
C++ g++	2.29	4,028	682	2.29	2% 5% 1% 100%

Ilustración 26. Go vs C++. Dígitos de Pi

En este caso el criterio para escoger Go sobre C++, entendiendo que este es un proyecto en el que además de los resultados se persigue un objetivo didáctico, aprender un nuevo lenguaje no enseñado durante la carrera, ofrece un innegable atractivo. Además, es un lenguaje que se encuentra en las primeras fases de desarrollo, lo que permite ver desde una perspectiva muy global, como se desarrolla un lenguaje poco a poco, comprendiendo mejor aún los mecanismos existentes bajo la lógica de los lenguajes de alto nivel.

4.2.2 Aperturas

Como ya se ha explicado anteriormente, existen numerosas aperturas utilizadas tanto por los grandes maestros como por jugadores menos experimentados.

Este gran número de aperturas derivan de unas aperturas estándar a las que se les van poco a poco añadiendo permutaciones y variaciones hasta llegar a tener cientos de miles de aperturas que difieren en un movimiento unas de otras.

Observando los requisitos definidos para el sistema y las necesidades de este, se observa que, una estructura que permita ejecutar movimientos rápidamente sin necesidad de hacer llamadas a la función de evaluación, o al algoritmo de búsqueda, sería de gran utilidad. Por ello, tener una biblioteca de aperturas lo más amplia posible, resulta de gran ayuda a la hora de comenzar la partida e ir desarrollando las máximas piezas posibles del tablero. Por esta razón, se ha decidido implementar una serie de estructuras en el programa, que permitan leer un fichero de aperturas y almacenarlo en memoria intermedia, para luego poder jugar de acuerdo a esas aperturas.

Sin embargo, al tomar esta decisión también se ha sopesado el gran inconveniente que surge con esta solución. Encontrar una fuente fiable, completa y con coeficientes de victorias de cada apertura, que además sea abierta, no resulta una tarea sencilla, y como se verá más adelante hay veces que hay elegir una lista de aperturas no tan completa para que cumpla los otros requisitos, y luego mediante arreglos en el programa simular las posibles permutaciones de cada apertura. Finalmente se ha decidido utilizar la fuente "PGNMentor", [PGN Mentor, 2008] ya que ofrece 2.281.761 aperturas, clasificadas según el movimiento de origen. Este número de aperturas es tan elevado que casi contempla cualquier variante de apertura posible.

4.2.3 Función de evaluación

Como se describió en el apartado 2, existen tres alternativas a la hora de elegir la función de evaluación que utilizará el programa. Además, existiría la posibilidad de crear una función de evaluación propia que decidiese el valor de cada movimiento propuesto. Sin embargo, esto plantearía vamos problemas, primero la dificultad evidente de crear una función de evaluación que fuese capaz de jugar a un juego tan complejo como el ajedrez. Anteriormente se vio, que todos los desarrolladores de motores de ajedrez eran jugadores profesionales de ajedrez a distintos niveles, con puntuaciones ELO superiores a los 2.000 puntos. Por lo que tener la capacidad de desarrollar un software capaz de acercarse a sus resultados supone un desafío enorme. En consecuencia, si se desarrollase una función propia que no fuese lo suficientemente buena, los resultados de los algoritmos de búsqueda quedarían condicionados, ya que el algoritmo de búsqueda y la función de evaluación están estrechamente relacionados. Por esta razón, se ha llegado a la conclusión de que la mejor opción, es escoger una función de evaluación ya implementada por un motor de ajedrez.

De entre los tres motores de ajedrez analizados, vimos que los tres mostraban resultados similares, sin embargo, aplicaban tácticas diferentes. De acuerdo a los requisitos establecidos, Stockfish es el que mejor se adapta a estos, debido a que tiene una gran fase final del juego, lo cual coincide con una gran parte de las pruebas que se van a realizar (se probarán retos de jaques famosos). Además, Stockfish no plantea tácticas que busquen las tablas, por lo que lo habitual es que termine encontrando una forma de llegar a realizar mate. Otro de los requisitos importantes, era que la función de evaluación fuese gratuita y open

source, para que fuese una función en constante mejora. Este requisito únicamente lo cumple Stockfish, por lo que este motor de ajedrez es el que mejor se adapta a las necesidades de este proyecto.

4.2.4 Algoritmo de búsqueda

Finalmente, se ha decidido escoger el algoritmo alfa-beta, buscando jugadas más agresivas que consigan estrategias con mayor probabilidad de victorias. Si bien es cierto, que con la poda alfa-beta existe más riesgo de perder la partida, por ir perdiendo piezas debido al efecto horizonte, este algoritmo resulta ganador en un mayor número de partidas que la búsqueda quiescente por sí sola. Puesto que el problema al que nos enfrentamos es un problema de suma cero es necesario escoger algoritmos que puedan aplicarse sobre este tipo de juegos, es decir, si bien es cierto que existen un gran número de algoritmos de búsqueda, se ha analizado el más usado actualmente (poda alfa-beta), y para tratar de ver el problema y posibles soluciones de este algoritmo, se ha analizado otro algoritmo existente.

En el apartado 2 se describieron los dos algoritmos de búsqueda más utilizados para búsqueda de IA en ajedrez. Sin embargo, no son las dos únicas opciones existentes. A la hora de elegir el algoritmo de búsqueda, al igual que con la función de evaluación se han barajado varias opciones de entre las existentes e incluso la opción de la creación de un algoritmo de búsqueda propio.

Con respecto a la creación de un nuevo algoritmo de búsqueda, y a diferencia de la función de evaluación, el principal problema no es la complejidad (que también es alta), sino los numerosos factores que hay que tener en cuenta, como el tiempo de desarrollo. Al final, se llegó a la conclusión, de que en conjunto el desarrollo de un nuevo algoritmo de búsqueda terminaría provocando que el proyecto se desviase de la planificación de tiempo inicial, y también se alejase de su propósito inicial. Por ello, se entendió que era más importante primero comprender como trabajan los algoritmos ya existentes, y plantear el desarrollo de un nuevo algoritmo como una línea futura de continuación de este proyecto.

De entre los algoritmos disponibles para analizar, se ha escogido la poda alfa-beta, debido a que es el algoritmo que en potencia expande menos nodos, es decir, el mejor caso y el caso de medio de la poda alfa-beta expanden menos nodos que algoritmos como Negamax o “Mate Search”. Además, el algoritmo escogido, tiene la ventaja frente a la búsqueda quiescente, que en el compromiso entre menor expansión de nodos y mejor elaboración de estrategia de juego, tiene un rendimiento claramente superior. Como ya se ha comentado, es cierto que la búsqueda quiescente puede llegar a expandir el mismo número de nodos a la misma profundidad, sin embargo, terminará elaborando una estrategia más conservadora, en la que buscará que no haya capturas en ninguno de los dos colores, lo cual provocaría que no se jugase la partida, o que como mucho la búsqueda quiescente se pasase la partida defendiéndose de los ataques del otro jugador.

4.2.5 Pruebas

Para probar el jugador de ajedrez hay distintas posibilidades. Existe la posibilidad de probarlo jugando directamente contra él, simplemente viendo los resultados que tendría el programa en un enfrentamiento contra un humano “normal”. Pero en este caso, los resultados dependerían de la habilidad del humano, y puesto que probarlo con un jugador profesional era complicado, se desechó esta posibilidad, ya que no se apreciarían realmente diferencias al ser partidas “sencillas” para la máquina.

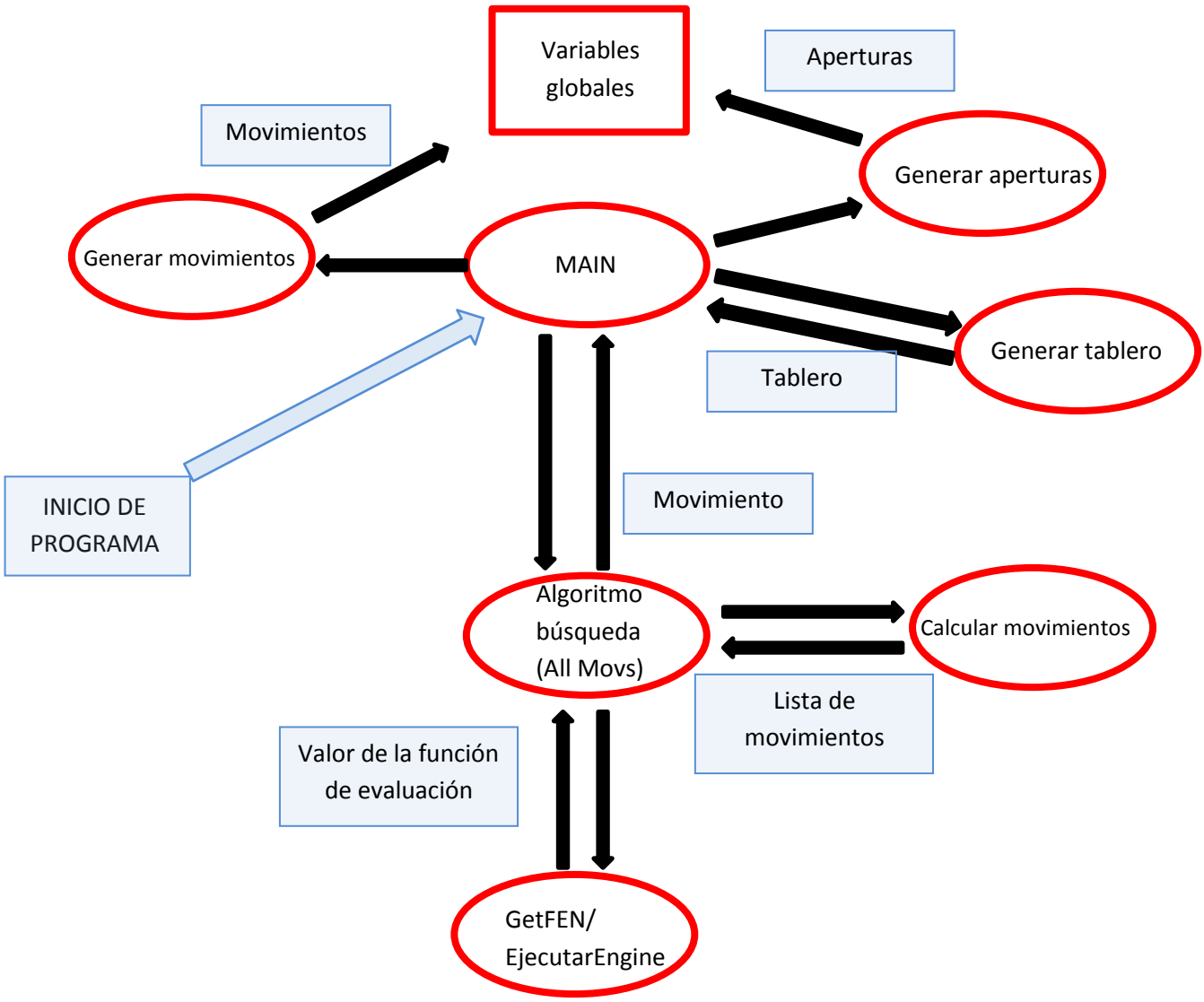
Otra posibilidad barajada fue conectar el programa a un servidor en el que mucha gente jugase al ajedrez, es decir, hacer que este programa jugase cientos de partidas y sacar los resultados generales de todas esas partidas. Sin embargo, esto requería cierto nivel de adaptación al software que incrementaba la complejidad, y sobre todo, el mayor problema, es que el programa no está preparado para jugar partidas blitz o rápidas de 3 ó 5 minutos, ya que el tiempo de ejecución por cada movimiento es demasiado elevado. Los resultados obtenidos si se conectase el programa a un servidor, donde otros muchos jugadores disputan partidas, sería que pierde todas las partidas al ser capaz de ejecutar sólo unos pocos movimientos por partidas. Así que, esta opción también quedó descartada.

La última opción posible y finalmente elegida fue, además de hacer pruebas para comprobar el correcto funcionamiento, conectar el programa de ajedrez con el motor de ajedrez de Stockfish. De tal forma que jugaría la función de Stockfish, con el algoritmo de búsqueda implementado y las aperturas dadas contra el

motor de ajedrez con la misma función de evaluación pero su propio motor de búsqueda y sus optimizaciones propias. Estas pruebas además, se harán a distintas profundidades, tanto por parte del jugador implementado durante el proyecto como por parte del motor de ajedrez Stockfish. Así se podrá ver, el rendimiento del sistema implementado frente a un sistema que es el considerado el segundo mejor jugador de ajedrez del mundo, y si supone alguna ventaja el sistema de aperturas diseñado.

4.3 Diseño del sistema

Para comprender adecuadamente el diseño del sistema a nivel software, hay comprender el diseño a nivel lógico. Para ello, se muestra a continuación un diagrama de flujo del sistema:



La primera funcionalidad que va a implementar el programa es tener almacenado el estado del tablero según se vayan sucediendo en este lo movimientos. Además se debe saber en cada momento que movimientos se pueden realizar. Para poder almacenar toda esa información se ha diseñado el sistema de la siguiente manera.

Para saber en todo momento que contiene cada casilla (pieza blanca, pieza negra o nada) se ha creado un array de 64 posiciones (una por cada casilla del

tablero). El índice del array va de 0 a 63, guardando como valor numérico el código de la pieza en esa casilla o en su defecto un 0. Este array se mantiene dentro del método main, que a su vez llama a los algoritmos de búsqueda para que le devuelvan el mejor movimiento. La tabla de valores de piezas en la siguiente:

Torre Blanca	**71**	**Torre Negra**	**-71**
Caballo Blanco	72	Caballo Negro	-72
Alfil Blanco	73	Alfil Negro	-73
Dama Blanca	74	Dama Negra	-74
Rey Blanco	75	Rey Negro	-75
Peón Blanco	76	Peón Negro	-76

Tabla 33. Valores posibles para el array de tablero

Como podemos observar para las piezas blancas se les asigna un valor positivo y para las piezas negras un valor negativo, este será de gran ayuda más adelante en el código para saber si la pieza posicionada en una casilla es del mismo color o no que la pieza con la que se compare. La distribución de casillas en el tablero se realiza de la siguiente manera. Dentro de cada casilla se puede ver el índice asignado.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Tabla 34. Valores tablero inicial

De tal forma que teniendo la disposición inicial de un tablero como este:

Ilustración 27. Tablero inicial

Tendríamos un array del tablero relleno de la siguiente manera:

TABLERO[64] = [-71] [-72] [-73] [-74] [-75] [-73] [-72] [-71]
[-76] [-76] [-76] [-76] [-76] [-76] [-76] [-76]
[0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [0] [0] [0] [0] [0] [0]
[0] [0] [0] [0] [0] [0] [0] [0]
[76] [76] [76] [76] [76] [76] [76] [76]
[71] [72] [73] [74] [75] [73] [72] [71]

Para saber en qué casilla y qué movimientos tiene disponibles cada una de las piezas, se crea ad-hoc una lista por pieza, de tal forma que existen 32 posibles listas, que en la primera posición de la lista contiene la casilla actual de la pieza y en las siguientes contiene los movimientos disponibles de esta. Cada nodo de la lista contiene como valor numérico la casilla a la que se puede desplazar esa pieza, y como muchas piezas se pueden desplazar en direcciones (torre, dama y alfil), entre cada sucesión de casilla de una dirección se ha incluido un nodo cuyo valor es 64 (valor nunca alcanzable en el tablero ya que va de 0-63). Este es el ejemplo de la lista que tendría una torre que se encuentra en el centro del tablero.

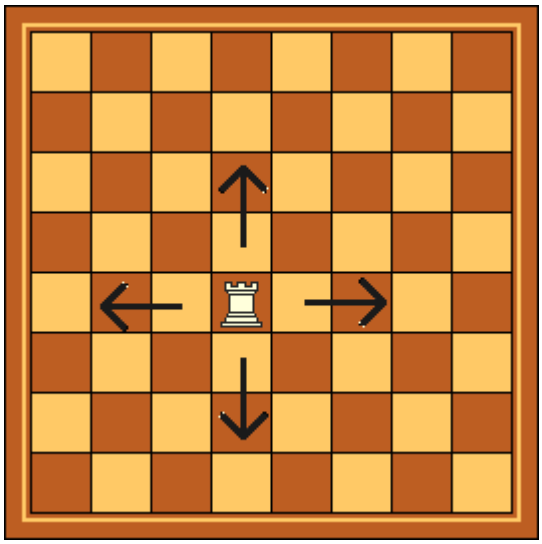
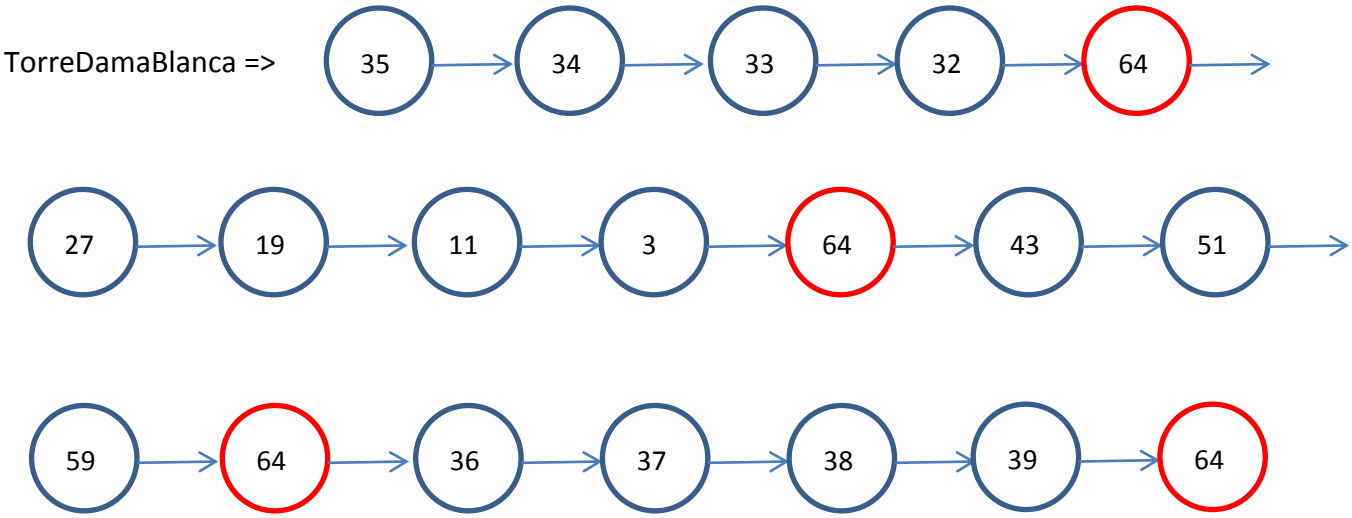


Ilustración 28. Movimientos disponibles de torre



Como podemos observar, cada vez que se terminan de incluir todas las casillas de una dirección (en este caso fila o columna) se inserta un 64 para indicar este final. Esta lista, una vez que se han obtenido todos los movimientos disponibles, es acortada con los movimientos que realmente son legales, es decir, se le eliminan aquellas posiciones inalcanzables porque haya una pieza interpuesta. También se eliminan los nodos con valor 64, ya que resultan innecesarios con la

lista ya acortada. En caso de que esa pieza sea del mismo color se elimina la casilla en la que se haya la pieza y todas las demás de esa dirección, en caso de que sea de distinto color se eliminan sólo el resto de casillas de la dirección. En este ejemplo vemos la torre anterior tapada por dos piezas, una de su mismo color y otra de distinto color.

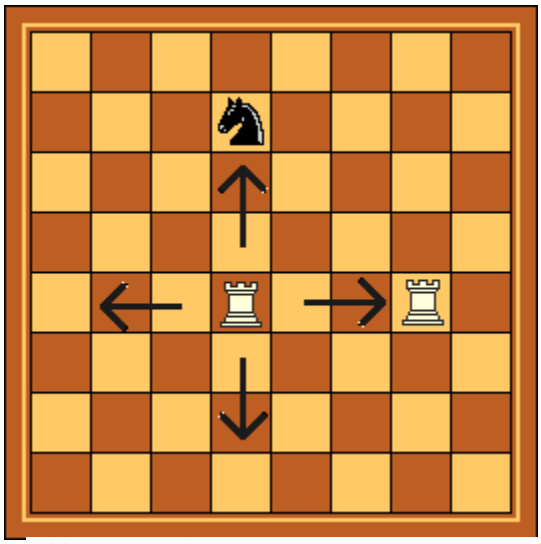
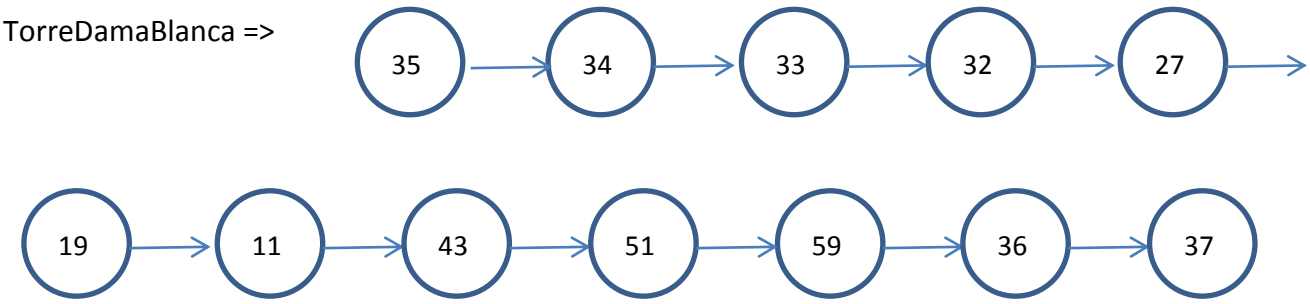


Tabla 35. Ejemplo acortar con piezas

La nueva lista quedaría:

TorreDamaBlanca =>

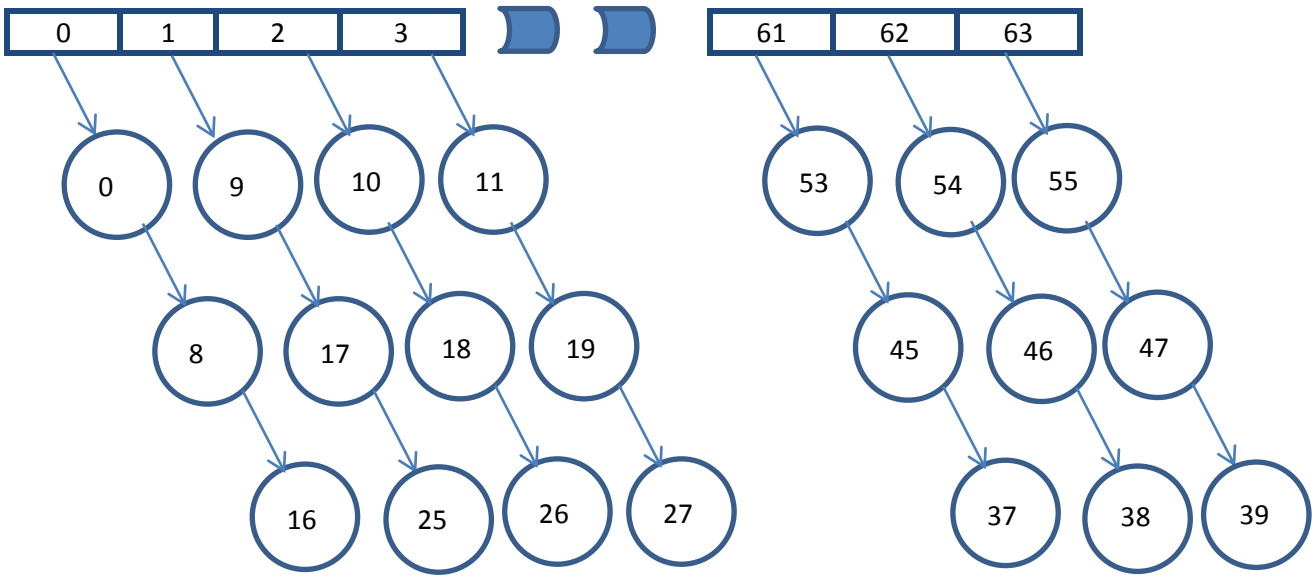


De esta manera cada vez que realizásemos el proceso de acortar todas las listas según los movimientos realmente disponibles, nos quedarían 16 listas (una por cada pieza disponible para mover en ese momento). Estas listas estarían ordenadas de función de la pieza a la que representan y contendrían todas las casillas a las que se puede mover esa pieza, es decir, uniendo las 16 listas, tendríamos a nivel lógico una única lista con todos los movimientos disponibles. En ese momento simplemente se trataría de llamar al árbol de búsqueda (que a su vez llama la función de evaluación), para que este nos devolviese el movimiento que se debe realizar a continuación.

Esta lista de movimientos disponibles es devuelta por un método, de tal forma, que cada vez que se necesita saber los movimientos disponibles en una casilla, simplemente se llama a ese método pasándole como parámetro la casilla y la disposición del tablero inicial.

Existe para cada pieza (independientemente del color o la posición), un método que almacena todos los movimientos disponibles de esa pieza en cada una de las 64 casillas del tablero. Esto se almacena en un array de 64 posiciones y en cada posición se almacena una lista. De tal forma, que cuando se necesite saber la lista de posiciones de una pieza sólo hay que hacer la llamada al método de obtener la lista en la posición adecuada. Los métodos de generación de movimientos por pieza, se llaman una sola vez al inicio del programa y estos guardan los datos en memoria intermedia, de tal forma que acceder a esos datos resulta bastante rápido. La estructura de datos que se almacena en memoria sigue este esquema.

Torre:



Una vez el explicado el diseño lógico de cómo se representa el tablero y las piezas, pasamos a explicar el diseño utilizado para implementar el sistema. El lenguaje Go, no es orientado a objeto por lo que no se pueden crear clases para organizar el código. Sin embargo, sí que se pueden crear paquetes que realizan una función similar. En este caso, hemos dividido el código en cinco paquetes relacionados entre ellos, tal y como puede verse en la figura.

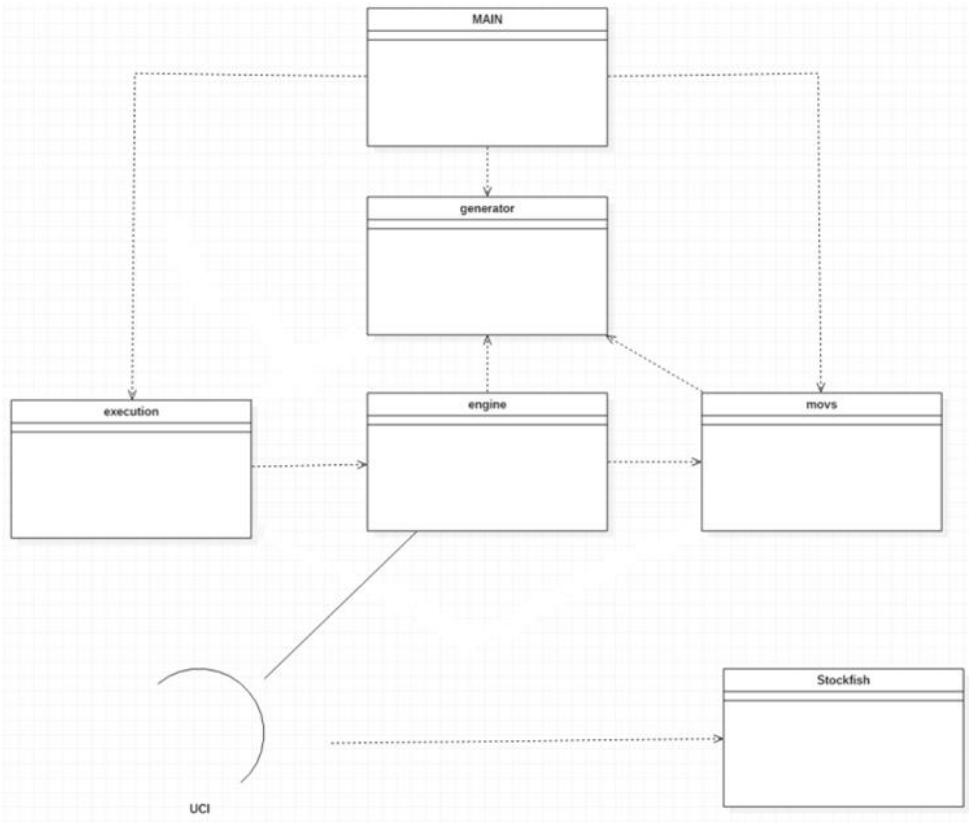


Ilustración 30. Diseño del sistema

A continuación se explica la función de cada paquete y se justifica su existencia.

Paquete generator: Este es el paquete central, en torno al que se articula el resto del sistema. En este paquete encontramos todas las variables globales que utiliza el sistema para almacenar el estado del tablero y de las piezas. Es mejor tener todas las variables en un solo paquete para así, cuando se quiera acceder a ellas, sólo se necesita importar este paquete. Como podemos observar, el resto de paquetes dependen de este, sin embargo, este paquete no depende de

ninguno, por lo que podemos decir que es el que contiene la funcionalidad más básica de todo el sistema.

Este paquete tiene los métodos encargados de generar todos los posibles movimientos de todas las piezas y luego almacenarlos en sus variables correspondientes. Además este paquete es el que declara las listas que luego van a usar el resto de paquetes para manejar los movimientos de cada pieza. También este paquete se encarga de leer el tablero pasado como argumento, y en función de eso rellenar la variable del tablero que luego permitirá saber el estado de cada casilla.

Paquete movs: Este paquete se encarga de manejar todos los métodos relacionados con los movimientos de las piezas. Este paquete depende únicamente del paquete generator, ya que necesita importar las variables de las piezas y utiliza los métodos de obtención de movimientos.

Este paquete se encarga de acortar las listas de las piezas de acuerdo a las piezas que tiene alrededor. También se encarga de calcular los movimientos disponibles, a través de enlazar llamadas a distintos métodos, de una pieza que se le pase como argumento.

Paquete execution: Este paquete importa varios paquetes. Utiliza generator para el acceso a las variables globales, movs para tener acceso a los métodos de acortar movimientos y el paquete engine para realizar llamadas a la función de evaluación. Este paquete se encarga de la ejecución de los movimientos

disponibles, por ello requiere hacer llamadas tanto a la función de evaluación como a los métodos de acortar listas. Además este paquete es el que contiene el método con todos los movimientos disponibles para ejecutar el algoritmo de búsqueda.

Paquete engine: Este paquete importa el paquete generator, igual que los anteriores, para tener acceso a las variables globales de las piezas. También importa el paquete movs para poder hacer comprobaciones sobre los movimientos.

Este es el paquete encargado de comunicarse con el motor de ajedrez y acceder a su función de evaluación. Los otros paquetes ejecutan llamadas a este método de tal forma que engine devuelva el valor de la función de evaluación para ese movimiento.

Paquete MAIN: Este es el paquete principal del sistema, que maneja el resto de paquetes e importa a todos los demás excepto el paquete engine. La funcionalidad principal de este paquete es recibir los parámetros con los que se ejecuta el programa y coordinar las llamadas a los métodos para terminar devolviéndole al usuario el movimiento ejecutado por el sistema y pedirle el siguiente movimiento disponible.

Interfaz UCI y Stockfish: Como ya se ha comentado, el motor de ajedrez elegido para llamar a su función de evaluación es Stockfish, pero para poder acceder a su función de evaluación se necesita hacer una llamada a esta función, ya que

Stockfish ofrece muchas funcionalidades. Para realizar esta llamada existe una interfaz UCI que es estándar para la comunicación con los motores de ajedrez. Para realizar esta llamada, sólo se necesita pasar un string FEN y llamar al ejecutable del motor de ajedrez. Esta interfaz además, permite hace que el programador no tenga que preocuparse de la diferencia de lenguajes, por ejemplo, en este caso, el programa de ajedrez está escrito en Go, y el motor de ajedrez en C++, pero la llamada se hace sin que eso influya.

Una vez explicado el diseño del sistema realizado pasamos a ver en cada paquete los métodos que existen y cómo funcionan.

4.3.1 Generator

En el paquete generator encontramos la clase generator.go con las siguientes variables globales y los siguientes métodos.

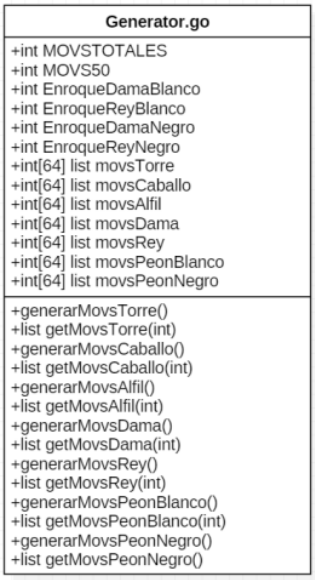


Ilustración 31. Esquema Generator.go

Esta clase es la encargada de generar las estructuras básicas, como ya se ha comentado. Declara las variables globales que utilizará todo el sistema.

MOVSTOTALES lleva la cuenta de turnos totales realizados hasta el momento y MOVS50 lleva el número de movimientos desde el último movimiento de peón o captura de pieza. Ambas variables son necesarias para construir los argumentos que se le deben pasar al motor de ajedrez.

Las variables de enroque sirven para saber si se ha producido un movimiento que inhabilite el enroque.

Las variables de movimientos almacenan todos los movimientos posibles en cualquiera de las casillas del tablero.

Los métodos de generar movimientos de cada pieza en cada casilla, recorren todas las casillas del tablero, y para cada casilla, en función de la pieza que se esté calculando, simulando el movimiento de esa pieza, van almacenando casillas disponibles. Estos métodos rellenan el array correspondiente, luego los métodos get reciben como parámetro una casilla y devuelven la lista de movimientos de esa pieza en esa casilla. Algunos métodos no requieren que se generen los movimientos de esa casilla expresamente, ya que se pueden construir como combinaciones de otros movimientos. Esta tabla indica cómo se generan los movimientos de cada pieza.

Movimientos Torre	Se calculan expresamente
Movimientos Alfil	Se calculan expresamente
Movimientos Caballo	Se calculan expresamente
Movimientos Dama	Son la combinación de los movimientos del alfil y la torre
Movimientos Rey	Se calculan expresamente
Movimientos Peón	Se calculan expresamente
Casillas Afectas	Se calculan como la combinación de la dama y el caballo

Tabla 36. Explicación de los cálculos por piezas

4.3.2 Movs

El paquete Movs, está formado por la clase movs.go que tiene los siguientes métodos.

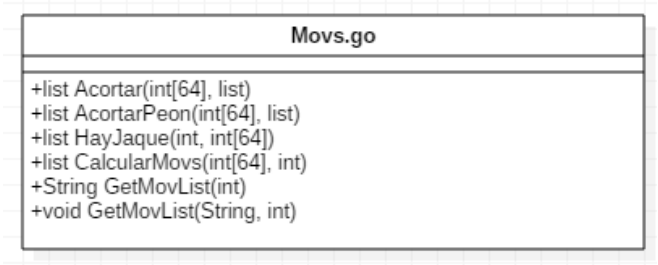


Ilustración 32. Esquema movs.go

Esta clase es la encargada de realizar las operaciones relacionadas con los movimientos de las piezas. El método acortar recibe una disposición de un tablero y la lista que tiene que acortar, luego devuelve otra lista en la que sólo se encuentran los movimientos legales, es decir, acorta la lista de movimientos en función de las piezas alrededor de la pieza a la cual pertenece la lista pasada

como argumento. Para hacer esto empieza leyendo la primera posición de la lista (que recordemos que es la casilla en la que se encuentra la pieza actualmente), y saca de ella la pieza a la que pertenece la lista. Después empieza a recorrer la lista viendo si las casillas están ocupadas o no, para ello compara la pieza con la variable tablero. En caso de encontrar una pieza del mismo color que la evaluada, borraría la casilla de la pieza y seguiría borrando hasta llegar al final de esa dirección (borrando también el nodo con valor a 64). Una vez llegado al final de la dirección reiniciaría todos los parámetros para empezar a borrar otra dirección. Cuando se habla de borrar nodos no es del todo correcto, porque este método lo que hace es copiar los nodos válidos a una lista paralela y devolver esa lista con los nodos válidos ya copiados. Cuando recorre las listas, y se encuentra con una lista cuya pieza es una torre realiza un tratamiento especial, ya que se debe ver si la torre tiene la capacidad o no de enrocar. Para ello, se comprueban las tres normas del enroque: no haber movido torre ni rey (esto se comprueba mediante la variable global enroqueXXXCOLOR), tener todas las casillas entre la torre y el rey disponibles (esto se comprueba al recorrer la lista) y que no haya jaque en las posiciones que recorra el rey (esto se comprueba mediante el método HayJaque). Una vez recorrida la lista y copiados los nodos a la lista final, se devuelve la lista final. De tal forma que el método que llame a Acortar no nota el cambio de lista, ya que ambas tienen una estructura similar.

El método de AcortarPeon sigue la misma lógica que el método anterior, pero hay que hacer un tratamiento algo distinto, ya que el peón puede realizar captura al paso y las capturas de los peones se hacen en diagonal. Para ello se comprueba si el peon puede moverse de primeras dos casillas y luego se

compruebas las casillas en diagonal para ver si existe la posibilidad de hacer una captura. Luego, se devuelve la lista con los nodos válidos.

El método HayJaque se encarga de decirle al método que lo invoca si existe jaque en la posición del enroque pasada por parámetro. Este método tiene un int que le indica en qué posición de enroque debe comprobar el jaque.

0	Enroque Dama Blanco (enroque largo)
1	Enroque Rey Blanco (enroque corto)
2	Enroque Dama Negro (enroque largo)
3	Enroque Rey Negro (enroque corto)

Tabla 37. Explicación de los enroques

Una vez que sabe en qué posición comprobar el enroque, comenzamos a comprobar las casillas adyacentes a las posiciones de jaque para ver si realmente alguna está en jaque. En el momento que se encuentre una, se devuelve true, porque con que haya una pieza haciendo jaque el resultado ya va a ser true. En caso de que no se encuentre ninguna pieza que amenace las casillas del enroque se devuelve false.

El método CalcularMovs recibe una casilla y un tablero como parámetro, y de esa casilla saca la pieza que tiene que modificar, es decir, saca la lista que tiene que modificar. Una vez localizada la lista que tiene que modificar, saca todos sus movimientos llamando al método Get y acorta todos sus movimientos llamando al método Acortar.

El método de GetMovList permite obtener el código del siguiente movimiento de la lista de aperturas. Para ello, recorre el array de lista de aperturas y busca aquellas aperturas que todavía estén disponibles (con un 1 en su posición inicial), y devuelve aquella apertura que sea más larga, para así buscar la estrategia, de mantener la fase de aperturas del juego el mayor tiempo posible.

El método SetMovList, recibe un código de un movimiento de apertura, y comprueba que posiciones del array de listas disponibles, contienen ese movimiento, para que en caso de que una lista disponible no contenga ese movimiento, cambiar el estado de esa lista a no disponible (marca el nodo inicial de la lista con un 0).

4.3.3 Execution

Dentro del paquete execution tenemos la clase Execution.go, encargada de gestionar todas las ejecuciones de movimientos (incluyendo el árbol de búsqueda).

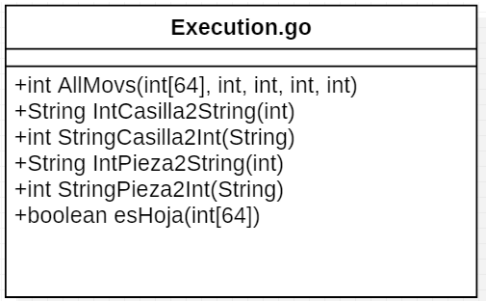


Ilustración 33. Esquema execution.go

El método AllMovs, es uno de los métodos principales del sistema por varias razones, primero porque es el método que lee de forma consecutiva todas las listas de todas las piezas del tablero, por lo que lee todos los movimientos disponibles a evaluar. Segundo, este método es el encargado de ejecutar el algoritmo de la poda alfa-beta, ya que cada vez que se evalúa una posición se hace una llamada recursiva a sí mismo para buscar en profundidad.

Este método, comprueba si el movimiento a ejecutar es un nodo hoja o ha alcanzado ya la profundidad máxima indicada por teclado, en ese caso devuelve la evaluación de la heurística en ese nodo. En caso de que no sea un nodo terminal, comprueba a que jugador le toca mover, y en función de eso ejecuta. Una vez dentro del jugador al que le toca mover, comprueba la lista de todas aquellas piezas que no hayan sido eliminadas del tablero, recorriendo uno a uno todos sus movimientos y haciéndose llamadas recursivas a sí mismo. De esta forma, está recorriendo el árbol con el sistema primero en profundidad. Gracias a esto, llega a un nodo terminal y comprueba su resultado en la función heurística, si le interesa seguir evaluando porque hay posibilidades de llegar a un máximo mayor que el máximo local, sigue expandiendo esa rama, en caso de no poder superar el máximo local hace un salto y poda esa rama, aunque sigue recorriendo el árbol, hasta recorrerlo o podarlo entero.

Los métodos de IntCasilla2String, StringCasilla2Int, IntPieza2String y StringPieza2Int, son utilizados por otros métodos para pasar de la codificación de usa a la máquina a una codificación más entendible por el humano, por ejemplo, como ya se ha explicado, existe un código para cada pieza, sin embargo, este código no es entendible por un ser humano, por ello, es método

muestra el nombre de la pieza, algo fácilmente entendible. Lo mismo sucede a la hora de indicarle a al programa que movimiento se quiere hacer. El jugador introducirá el nombre de la pieza y su color, y es necesario transformar ese nombre en el código que usa la máquina para representar esa pieza. Así que, estos métodos únicamente

El método esHoja, es un método privado de esta clase, que utiliza el método que ejecuta el árbol de búsqueda, y comprueba si el nodo pasado por parámetro es un nodo terminal. En el ajedrez la partida termina cuando se hace jaque mate o se llega a tablas, a través de la regla de los 50 movimientos, y eso es precisamente lo que comprueba este método.

4.3.4 Engine

El paquete engine, contiene la clase Engine.go. Esta clase es la encargada de comunicarse con el motor de ajedrez para ejecutar la función de evaluación de este.

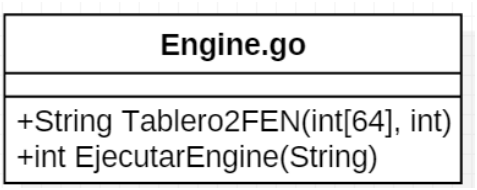


Ilustración 34. Esquema engine.go

El método Tablero2FEN, es el encargado de construir el String fen. Este string es el que le indica al motor de ajedrez la situación del tablero, el turno de movimiento, los posibles enroques, las capturas de peón al paso, los

movimientos según la regla de los 50 movimientos y los movimientos totales. Para construir este string, primero se lee el tablero actual modificado según el movimiento pasado por parámetro. Después se le copia el turno de movimiento. Lo siguiente que se hace, es comprobar la disponibilidad de los enroques y copiar el código correspondiente para cada uno de los enroques. Por último se le copia el código de la captura al paso en caso de que la hubiese y el número total de movimientos desde la última captura o movimiento de peón, y el número total de turnos hasta el momento.

El método EjecutarEngine, es el encargado de comunicarse con el motor de ajedrez. Recibe como parámetro el fen y crea las estructuras adecuadas para realizar la invocación del ejecutable del motor de ajedrez. Una vez llamado al motor de ajedrez, este le devuelve una valor correspondiente ejecutar la función de evaluación según esa disposición del tablero. A continuación, saca de toda la información recibida el valor numérico de la evaluación de ese movimiento y devuelve este valor.

4.3.5 Main

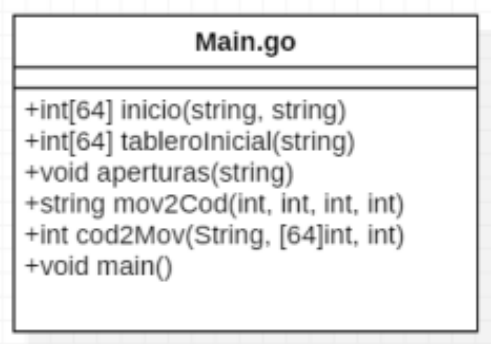


Ilustración 35. Esquema main.go

Este paquete contiene la clase Main.go. Esta es la clase encargada de realizar todo el manejo de la aplicación, leyendo por teclado los movimientos, ejecutando los métodos para calcular el siguiente movimiento, e imprimiendo el movimiento recientemente ejecutado por la máquina, para así volver a pedir otro movimiento al usuario y volver a empezar el ciclo.

El método Inicio, se encarga de inicializar las variables que se van a utilizar durante la ejecución de todo el sistema. Además, llama a los métodos necesarios para rellenar en memoria intermedia todas las estructuras con los movimientos disponibles de cada pieza. También hace una llamada al método de tableroInicial, para obtener el tablero leído por parámetro, y hace otra llamada al método de aperturas, para leer las aperturas pasadas por parámetro.

El método tableroInicial, se encarga de leer línea a línea el fichero de texto en el que se encuentra el tablero marcado como inicial. Luego, va rellenando la variable tablero que devolverá para indicarle al método inicio, cual es la disposición del tablero desde la que se va a empezar a jugar.

El método aperturas, va leyendo cada línea del fichero pasado como parámetro. Cada una de estas líneas contiene los datos de una apertura, que va almacenando en memoria intermedia, para cuando necesiten ser consultados por el sistema.

Los métodos cod2Mov y mov2Cod son los encargados de procesar el string de código recibido desde la lista de aperturas y convertirlo en un movimiento y viceversa.

El método main, es el encargado de generar de llamar al método de inicio e ir imprimiendo y leyendo todos los movimientos realizados tanto por la máquina como por el usuario. Para ello, ejecuta un bucle cuya única condición es que si la variable de fin de partida se activa, se termina el bucle dando por ganador al jugador que haya realizado el mate. Dentro del bucle lee la pieza que quiere mover el usuario y las casillas inicial y final, comprueba si es válido y llama hace la llamada al sistema para que el programa calcule el siguiente movimiento, y así ir jugando poco a poco la partida.

Capítulo 5.

Pruebas y resultados

En este capítulo, se procede a realizar las pruebas anteriormente comentadas y explicar los resultados obtenidos en cada una de ellas. En todos los casos probados, se ejecutará el algoritmo de búsqueda poda alfa-beta y/o las aperturas frente al motor de ajedrez Stockfish, a distintas profundidades. Además se detallarán las conclusiones sacadas en torno a estos resultados.

Las pruebas se van a realizar sobre un ordenador con un procesador Inter Core i5 M430 (4 núcleos a 2.27 GHz), con 4 GB de RAM DDR3 a 1067 MHz.

Durante las pruebas se analizarán tres parámetros, en torno a los cuales, se observarán y comentarán las diversas conclusiones obtenidas. Estos parámetros son:

- **Nodos totales:** Estos son los nodos expandidos por el algoritmo de búsqueda, incluye tanto los nodos hoja como lo no hoja. Este parámetro permite comprobar cuantas posibilidades necesita comprobar el algoritmo para encontrar la solución.
- **Nodos hoja:** Este parámetro contempla los nodos hoja expandidos por el algoritmo de búsqueda, es decir, aquellos nodos en los que no se hace llamadas a la función de evaluación.
- **Tiempo:** Este parámetro mide el tiempo empleado por el sistema para resolver el problema planteado. En ocasiones puede hacer referencia al tiempo total, y en otras al tiempo medio por cada turno o movimiento, en cada caso se especificará a qué hace referencia.

Para realizar las pruebas, lo que se va a hacer, es que el jugador de blancas será quien ejecute la función de evaluación más el algoritmo de búsqueda, y el

jugador de negras, ejecutará el motor de ajedrez de Stockfish a la misma profundidad que el jugador de blancas. Los datos tomados serán los correspondientes al jugador de blancas (sistema desarrollado).

5.1 Nodos totales

Analizar este parámetro es de suma importancia, ya que el número de nodos expandidos por el algoritmo de búsqueda, será lo que determine finalmente tanto el tiempo empleado por movimiento, como la capacidad del sistema para realizar una buena jugada. Además, ver la progresión de la expansión de nodos según se va incrementado la profundidad de búsqueda, resulta muy útil para saber cómo se comporta el algoritmo. En este caso, se va indicar en cada profundidad la media de nodos expandidos en cada movimiento, es decir, se sumarán todos los nodos expandidos durante la partida y se dividirán entre el número de movimientos.

Profundidad	1	2	3	4	5	6
Poda alfa-beta	27.12	568.47	3.574,41	5.495,94	128.419,71	716.413,50

Tabla 38. Resultados de los nodos totales

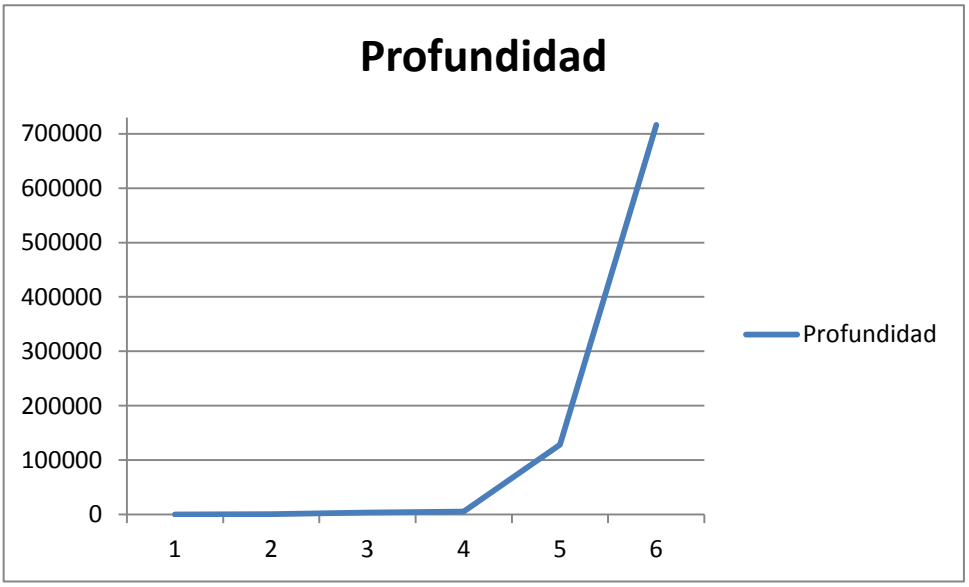


Ilustración 36. Gráfico de los nodos totales

5.2 Nodos hoja

Este parámetro, indica el número de nodos hoja expandidos, es decir, aquellos nodos, en los que el algoritmo de búsqueda ha dejado de ejecutarse, bien por haber llegado a la profundidad máxima, bien por haberse producido el final de la partida en esa rama de la búsqueda. Este dato, permite conocer el número de llamadas a la función de evaluación de Stockfish, y puesto que se ha visto que estas llamadas son las causantes del bajo rendimiento del sistema, saber cuántas se hace aporta información valiosa. Igual que en el caso anterior, se ofrece una media de llamadas por movimientos ejecutados.

Profundidad	1	2	3	4	5	6
Poda alfa-beta	27,12	502,84	2.459,81	2.842,14	92.086,03	431.873,45

Tabla 39. Valores de los nodos hoja

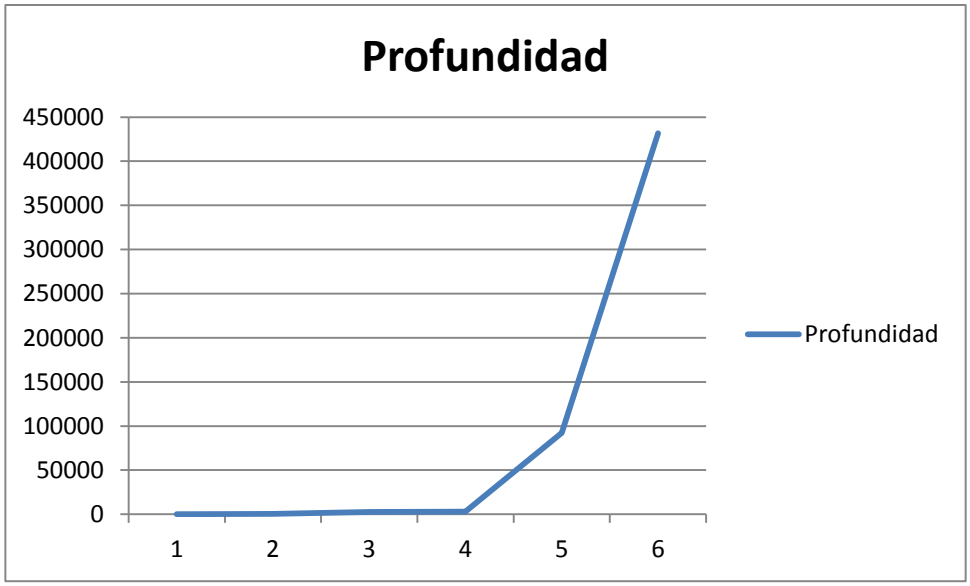


Ilustración 37. Gráfico de los nodos hoja

5.3 Tiempo

Este parámetro, indica el tiempo en segundos empleado en ejecutar movimientos por el sistema. Igual que en los casos anteriores, se toma una media de tiempos entre los movimientos ejecutados. Este parámetro permite conocer cómo es de bueno o de malo el sistema en una ejecución real, ya que cuando se juega al ajedrez, no se pueden apreciar los nodos expandidos por un sistema (nodos expandidos son parámetros para los desarrolladores), sino que lo que aprecian los jugadores, es el tiempo tardado.

Para evaluar el tiempo, se van a realizar varias pruebas, ya que en este apartado pueden entrar en juego las aperturas. Primero se va a probar el sistema sin aperturas jugando contra el sistema con aperturas a distintas profundidades. Luego, se va a probar el sistema sin aperturas jugando contra el motor de ajedrez Stockfish. Y finalmente, se realizará la pruebas más importante, que evalúa todo el sistema contra un motor de ajedrez, por lo que, se enfrentará al sistema con aperturas frente a Stockfish, todo ello a distintas profundidades.

Profundidad	1	2	3	4	5	6
Sistema SIN aperturas	1,28”	18,41”	94,75”	108,42”	582,30”	945,74”
Sistema CON aperturas	0.93”	15,63”	92,56”	107,39”	580,02”	944,09”

Tabla 40. Tiempo tardado SIN vs CON aperturas

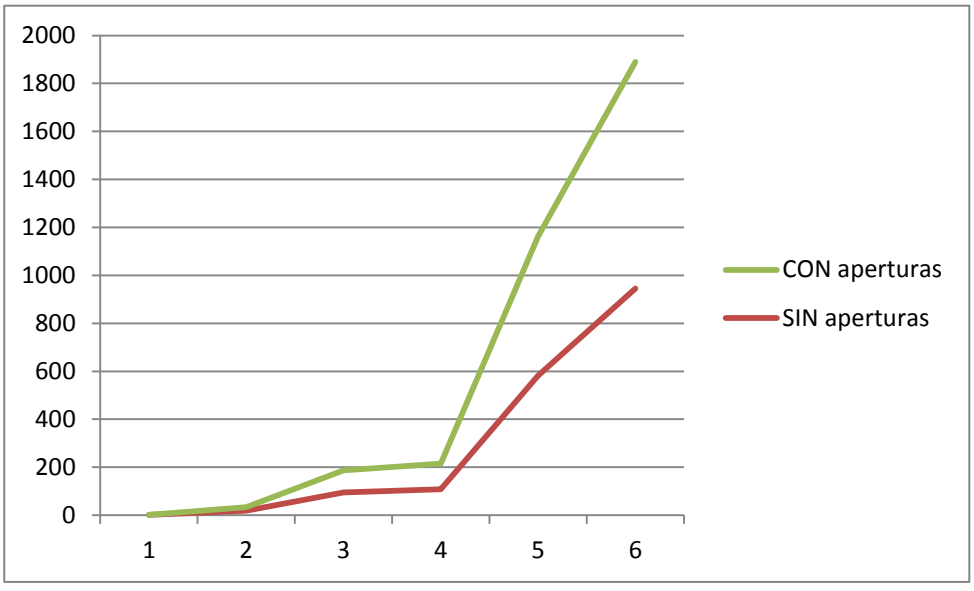


Ilustración 38. Gráfico del tiempo tardado SIN vs CON aperturas

Profundidad	1	2	3	4	5	6
Sistema SIN aperturas	1,33”	20,18”	100,18”	115,38”	590,84”	943,50”
Stockfish	1,20”	1,34”	1,81”	1,89”	2,02”	2,34”

Tabla 41. Tiempo tardado SIN aperturas vs Stockfish

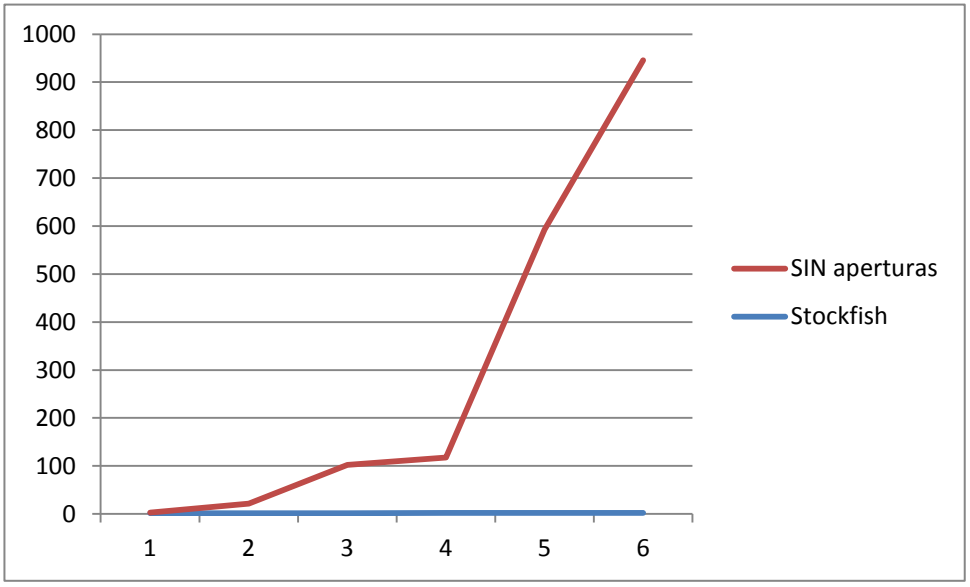


Tabla 42. Gráfico tiempo tardado SIN aperturas vs Stockfish

Profundidad	1	2	3	4	5	6
Sistema CON aperturas	1,20”	18,47”	92,15”	117,41”	580,65”	920,45”
Stockfish	1,17”	1,37”	1,82”	1,90”	2,35”	2,47”

Tabla 43. . Tiempo tardado CON aperturas vs Stockfish

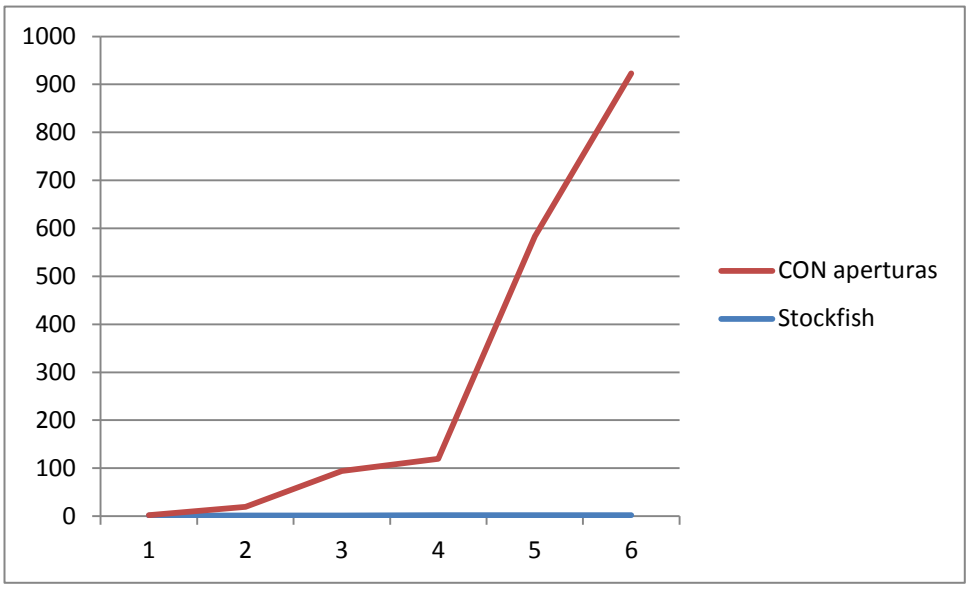


Ilustración 39. Gráfico del tiempo tardado CON aperturas vs Stockfish

5.4 Resultados

En este apartado se analizan los resultados obtenidos anteriormente. Indicando cuáles son las conclusiones obtenidas de las pruebas realizadas.

En todos los casos, como es lógico resulta vencedor el motor de ajedrez Stockfish, y en el caso de las pruebas del sistema con aperturas contra el sistema sin aperturas, resulta ganador en todos los casos el sistema con aperturas, sin embargo, se fue viendo, que cuando se ejecutaba el sistema con menos profundidad, el sistema con aperturas ganaba con más ventaja, por el contrario, cuando se usaba más profundidad, la diferencia entre ambos disminuía.

En el apartado del número de nodos totales expandidos por el sistema sin aperturas, el resultado fue que Stockfish ganó en todos los casos, sin embargo, en el primer caso la partida fue bastante igualada, yéndose a 59 movimientos. En el resto de los casos, no se pasó de los 30 movimientos. Esto es un resultado lógico, ya que el motor de ajedrez de Stockfish implementa muchas mejoras con respecto al sistema desarrollado en este proyecto. Por lo que la primera partida que se jugó a profundidad 1 no existe mucha diferencia entre las mejoras de Stockfish y el sistema, y por ello, el sistema pudo hacer frente al motor de ajedrez. En el resto de partidas, entraron en juego las diversas funcionalidades extras de Stockfish al ejecutar algoritmos de búsqueda y eso resultó determinante.

En el apartado de nodos hoja expandidos por el sistema, el resultado de la partida fue el mismo que el resultado en el apartado anterior (ya que fueron las mismas ejecuciones). Sin embargo, en este punto, podemos observar, el número

de llamadas hechas a la función de evaluación. Como vemos en las gráficas, el número de nodos expandidos y nodos hoja, crece de forma proporcional, lo cual es bastante lógico, además también vemos que el tiempo de ejecución del sistema sin nodos hoja, crece también de forma proporcional a los otros dos valores. Esto indica que casi todo el tiempo de ejecución empleado por el sistema, se utiliza en hacer las llamadas a la interfaz UCI, es decir, el sistema implementado no consume apenas tiempo, sin embargo, el sistema global es lento debido a las llamadas al UCI. Esto es un punto a tener en cuenta, a la hora de realizar mejoras sobre el sistema.

En cuanto a las pruebas de tiempo, se el sistema con aperturas ganó todas las partidas, aunque las partidas con mayores profundidades fueron más igualadas. La explicación de que ganase todas las partidas es sencilla, desde un principio pudo desarrollar una estrategia ya seguida por jugadores expertos, por lo que a igualdad de condiciones en el juego medio y el fin del juego, el sistema con aperturas, terminó haciendo buena, la ventaja obtenida durante la fase de aperturas. En cuanto al tiempo, aunque en un principio se podría pensar que existe una gran diferencia, hay que tener en cuenta, que en el momento en el que el sistema con aperturas deja de poder utilizarlas, los tiempos se igualan, y puesto que han sido partidas de unos 60 ó 70 movimientos, y las aperturas no han durado más de 15 movimientos, los tiempos durante el medio juego y el fin del juego se han compensado.

En el apartado del sistema sin aperturas frente a Stockfish, existe una gran diferencia, ya que Stockfish además de haber ganado todas las partidas, como ya se ha comentado, ha empleado mucho menos tiempo, siempre por debajo de

los 10 segundos. Esto supone una brecha de tiempo insalvable para el sistema desarrollado, en caso de enfrentarse a Stockfish en una partida real. Esta diferencia de tiempo, es fácilmente justificable debido a los mecanismos de optimización del motor de ajedrez, y también, por el hecho ya comentado, del que el sistema es excesivamente lento en las llamadas a través del UCI.

En el apartado del sistema con aperturas frente a Stockfish, se puede ver resultados similares a los anteriores, con la ligera diferencia, de que las partidas se han alargado más, en media se han jugado 10 movimientos más en este tipo de pruebas, por lo que se puede concluir, que realmente, el sistema de aperturas, sí que resulta beneficioso para el sistema desarrollado.

Por último, se han realizado dos prueba más, enfrentar el sistema sin aperturas con el sistema con aperturas pero utilizando profundidades distintas. Puesto que el sistema con aperturas ya ha vencido en las pruebas anteriores, se ha decidido aumentar la profundidad del sistema sin aperturas y ver en qué momento ganaba al sistema dotado de la biblioteca de aperturas. En esta tabla vemos los resultados.

	CON aperturas	SIN aperturas	Resultado
Profundidad	1	2	1-0
	1	3	1-0
	1	4	0-1
	2	3	1-0
	2	4	1-0
	2	5	0-1
	3	4	1-0
	3	5	0-1
	4	5	1-0
	4	6	0-1
	5	6	0-1
	6	6	1-0
	6	7	0-1

Tabla 44. Resultados CON vs SIN aperturas

Como se observa a partir de cierta profundidad, la estrategia de las aperturas empieza a perder efectividad, ya que mientras el sistema con aperturas está ejecutando aperturas el sistema contrario está ejecutando a la profundidad indicada, por ello movimientos más “elaborados”, aunque también mucho más lentos, ya que ejecutar una apertura siempre es menos de un segundo y ejecutar a profundidad 6 suponen unos 12 minutos.

Otra prueba que se ha realizado, es la de probar el sistema con aperturas a más profundidad que el motor de ajedrez Stockfish. En este caso, se pretendía ver, a qué profundidad mínima podía ganar el sistema utilizando Stockfish a profundidad 4 (una profundidad básica). Por razones de tiempo de ejecución, la profundidad máxima a la que se ha ejecutado el sistema ha sido a 7, que ha sido la primera vez que el sistema ha sido capaz de ganar.

Una última prueba realizada, ha sido ejecutar el sistema sin aperturas, es decir, con el sistema utilizando el algoritmo de búsqueda de forma constante con las blancas, y para las negras, hacer que usen el algoritmo de búsqueda, pero en vez de llamar a la función de evaluación cuando encuentre un nodo hoja, que devuelva un valor cualquiera sin tener que hacer llamadas. Con este se persigue ver cuánto tiempo dedica el sistema a calcular movimientos y cuánto a hacer las llamadas al UCI.

Profundidad	1	2	3	4	5	6
Sistema SIN aperturas	1,20”	18,47”	92,15”	117,41”	580,65”	920,45”
Sistema SIN función de evaluación	1,07”	1,97”	2,12”	3,80”	4,95”	6,17”

Tabla 45. Tiempos CON y SIN llamada a la función de evaluación

En este caso, no ilustran los resultados con una gráfica porque no quedarían realmente ilustrativos. Sin embargo, es evidente la diferencia que existe entre llamar o no al UCI. Esto demuestra que la parte lenta del sistema son las llamadas al UCI.

Como conclusión podríamos decir, que el sistema implementado es inferior a Stockfish principalmente debido al tiempo de llamar al UCI. Además, se ha visto que la mejora de la aplicación de una biblioteca de aperturas es muy beneficiosa.

Capítulo 6.
Conclusiones

En este capítulo se resumirán los principales resultados alcanzados, evaluando si se ajustan o no a los objetivos planteados inicialmente. También se mencionarán los problemas encontrados durante la realización de este proyecto y se destacarán los resultados obtenidos y conclusiones sacadas al finalizar todo el trabajo.

6.1 Objetivos

A continuación se listan los objetivos marcados inicialmente, valorando si se han cumplido o no y explicando el por qué.

- **Software en Go:** En este apartado se fijaba como objetivo, conseguir implementar un software capaz de jugar al ajedrez de acuerdo a las reglas internacionales. Este objetivo se ha cumplido, ya que el sistema tiene la capacidad de calcular todos los movimientos legales disponibles en todas las situaciones posibles. Además, se especificaba que esto se hiciese con el lenguaje de programación Go, que es precisamente el único lenguaje utilizado en todo el sistema.

- **Función de evaluación:** Este objetivo, indicaba que el sistema debía ser capaz de conectarse a la función de evaluación del motor de ajedrez Stockfish. Esto se ha conseguido, haciendo una llamada a Stockfish mediante una interfaz UCI, y procesando la respuesta para obtener el valor de la función de evaluación.

- **Algoritmo de búsqueda:** Este apartado especificaba que debía implementarse un algoritmo de búsqueda, que se ejecutarían sobre el sistema para encontrar el mejor movimiento. Este algoritmo, poda alfa-beta, además, debía ejecutarse a una profundidad mínima de 4. Este objetivo se ha cumplido, ya que el sistema utiliza este algoritmo cuando calcula el mejor movimiento. Sin embargo, y aunque no se especificase claramente, la intención era que el sistema ejecutase este algoritmo en un tiempo razonable, para jugar una partida con un humano y que esta partida no se ralentizase en exceso. Esta última parte no ha quedado perfectamente resuelta, ya que por diversos problemas más adelante mencionados, el sistema es más lento de lo esperado al ejecutar el árbol de búsqueda.

- **Análisis algoritmos:** Este objetivo marcaba que el sistema pudiese utilizarse para hacer sacar conclusiones de rendimientos del algoritmo de búsqueda comentado anteriormente. Para así, poder decidir cómo se ajusta al problema del juego del ajedrez. Este objetivo ha sido resulto, como se puede comprobar en el capítulo de pruebas.
- **Aperturas:** En este apartado, se pretendía que el sistema tuviese la capacidad de utilizar una lista de aperturas realizadas por otros usuarios, para reconocer y ejecutar esas aperturas. Como se ha explicado en el desarrollo, esto se ha conseguido descargando una lista considerablemente grande de aperturas, e implementando en el sistema

las estructuras de datos para almacenar en memoria intermedia estas aperturas y acceder a ellas rápidamente.

- **Conceptos:** Este objetivo marcaba, que el desarrollo del proyecto sirviese para adquirir conocimientos sobre algunos de algoritmos de búsqueda que se pueden implementar para hacer búsquedas en profundidad en base a una función de evaluación. Esto se ha conseguido, ya que para realizar el diseño y la implementación es absolutamente necesario, comprender todos los conceptos. Además, también se han visto los problemas que tienen cierto tipos de algoritmos como la poda alfa-beta y su posible solución, búsqueda quiescente.

Una vez repasados los objetivos propuestos y alcanzados, se puede mencionar también lo que supone la realización de este proyecto. El desarrollo, la implementación y la obtención de resultados del sistema realizada en este proyecto, supone una base muy importante de cómo trabaja ahora mismo la IA siguiendo funciones heurísticas. Aunque este caso, es totalmente aplicado al ajedrez, se observa que los algoritmos de búsqueda, son fácilmente extrapolables a la toma de decisiones en problemas reales. Por ello, este proyecto supone un punto de partida muy adecuado para aclarar conocimientos, y una base muy sólida para construir un sistema de IA, no sólo capaz de solucionar problemas en el espacio del ajedrez sino también en el mundo real. Además, este trabajo puede servir también para comprender y desarrollar un motor de ajedrez con heurística propia y algoritmo de búsqueda propio.

6.2 Problemas encontrados

En este apartado vamos a analizar los problemas encontrados durante el desarrollo de este proyecto. Estos problemas están relacionados con diversos aspectos del sistema, que en un principio se planificaron de acuerdo a un diseño, pero con el desarrollo del sistema se ha ido viendo que era necesario cambiarlos.

Uno de los puntos en los que ha habido más problemas, ha sido con respecto al lenguaje. Go es un lenguaje bastante nuevo, del que al comienzo del proyecto, no tenía ningún conocimiento, y aunque existe mucha documentación sobre él y es similar a otros lenguajes como java y c, tiene sus particularidades a la hora del manejo de datos o las estructuras de memoria. Además tiene su sintaxis propia, algo diferente a la de otros lenguajes más comunes, por ejemplo, se pueden declarar variables únicamente con el símbolo ‘:=’, no existe la sentencia ‘while’, las clases deben llamarse como los paquetes en los que se encuentran o, que un programa en Go, no compila si existe un warning de una variable declara y no utilizada. Por todo ello, adaptarse al manejo de las estructuras de Go y la sintaxis propia, ha sido un problema común a lo largo del proyecto, sin embargo, era algo esperado y absolutamente lógico cuando se maneja un nuevo lenguaje por primera vez.

Otro de los problemas surgidos durante el proyecto, ha sido la comunicación con la función de evaluación. Como se ya se ha comentado, el motor de ajedrez, Stockfish, está escrito en C++ y el código del sistema en Go. Por eso, para hacer llamadas de un código a otro se necesitaba una vía de comunicación. Durante

algún tiempo, se intentó utilizar SWIG, que proporciona una comunicación intermedia entre código escrito en C++ y código escrito en cualquier otro lenguaje. Sin embargo, como se ha comentado esto tenía varios problemas de implementación, por lo que se terminó descartando, por ello, se necesitaba una nueva solución. Como ya se ha visto, esa nueva solución fue la interfaz de comunicación UCI, que permite hacer la llamada a la función de evaluación.

El principal problema surgido durante la implementación del sistema y que no se ha podido resolver completamente, es el problema del rendimiento al ejecutar los algoritmos de búsqueda, ya que emplean un tiempo demasiado alto, que hace que no resulte cómodo jugar una partida entre un humano y el sistema. Este problema se debe a que el tiempo de ejecución de la función de evaluación es bastante más alto del esperado en un principio, y puesto que el algoritmo de búsqueda llama repetidas veces a esa función de evaluación, este árbol de búsqueda se vuelve más lento de lo esperado. Este problema se ha arreglado parcialmente con mejoras en el código, que hacen que el código de manejo de estructuras sea rápido y apenas pierda tiempo calculando movimientos. También el sistema de aperturas mejora esta situación, aunque no se solucione totalmente. Aquí encontramos una de las posibles mejoras del sistema que se puedan realizar en el futuro.

El último problema que hay que mencionar está relacionado con la lista de aperturas. Para conseguir una lista lo más completa posible, se había decidido crear un crawler que visitase una página sacando todas las aperturas disponibles. Sin embargo, una vez terminado el crawler, al ejecutarlo sobre la página en cuestión, tras 20 peticiones la página se caía. Esto puede ser debido a

las políticas de seguridad de este sitio web, que lo detecta como un ataque y prohíbe el acceso. Esto provocaba que no se pudiese sacar una lista de aperturas, para solucionarlo, se buscaron otras web en las que sólo hubiese que hacer una petición http, y de esa petición sacar todas las aperturas posible, y gracias a esta nueva estrategia se consiguió obtener una lista de aperturas lo suficientemente grande como para resultar útil para el sistema.

6.3 Resultados algoritmos

En función a los resultados obtenidos en el apartado de pruebas, podemos remarcar ciertas ideas. La primera, como ya se ha comentado, es que el mecanismo de tener una biblioteca de aperturas añadida al sistema favorece considerablemente su rendimiento. Primero, hace que durante una fase del juego, el sistema ejecute más rápido, aunque luego esa ventaja se vea neutralizada, y segundo y más importante, permite desarrollar una estrategia prácticamente definitiva, que hace que se llegue al medio juego con una ventaja suficiente como para que luego resulte definitiva. Esto se debe principalmente, porque a través de las aperturas se desarrollan todas las piezas, con lo cual cuando entra en funcionamiento la función de evaluación y el algoritmo de búsqueda, todo el sistema resulta más eficaz.

Otro punto observado en las pruebas, es que el sistema implementado queda muy lejos de un motor de ajedrez como puede ser Stockfish, debido al tiempo empleado en ejecución y a la diferencia de resultados en las partidas, es decir, Stockfish resulta ganador en todas ellas. Viendo los resultados obtenidos,

también se debe destacar el principal fallo cometido en este proyecto, no haberse dado cuenta del tiempo que consume la llamada al UCI, que hace que tal y como está implementado, sea inviable usar el sistema para jugar contra un humano, debido a que a profundidad 6 (un gran maestro llega a valorar jugadas a mayor profundidad), ya tarda más de 15 minutos por jugada, lo que desemboca en partidas muy largas y pesadas para un jugador humano.

Por lo tanto, si se infiere algún conocimiento de las pruebas, se puede ver claramente, que la parte que lastra el tiempo de ejecución de todo el sistema es el UCI, ya que el resto del sistema resulta bastante eficiente en ejecución. Gracias a esto, se puede desarrollar una nueva línea de desarrollo en la que se busque posibles alternativas al uso del UCI, que aceleren el sistema de forma global y lo hagan adecuado para ser usado contra un ser humano.

Capítulo 7.

Líneas futuras

En este capítulo, se explican las posibles formas de continuar este proyecto, ya que como se ha explicado, pueden servir de punto de partida para proyecto mucho más grandes.

7.1 Algoritmo de búsqueda

Durante todo el proyecto, se ha visto qué necesidades tiene un algoritmo de búsqueda aplicado a este problema, además de ver sus posibles características y cómo influyen en el resultado y los posibles problemas que pueden surgir derivados de escoger en cada caso un algoritmo u otro.

Una posible forma de continuar este proyecto, es en función de los resultados obtenidos en las pruebas, buscar una forma de mejor los algoritmos ya existentes, haciendo que encuentra la solución a más profundidad, ya sea expandiendo menos nodos, o en caso de expandir el mismo número, expandir sólo aquellos que realmente sean necesarios. También podría buscarse desarrollar mejoras para que estos árboles de búsquedas diseñen estrategias más efectivas a la hora de jugar una partida (consigan más victorias).

Por otro lado, puede trabajarse en la misma dirección pero de otra manera. Este trabajo podría usarse para conocer la base de los algoritmos de búsqueda que se está utilizando ahora mismo y construir un algoritmo totalmente nuevo, que aprovechando el conocimiento que ya se tiene, realice una búsqueda más efectiva (gane más partidas) o más eficiente (lo haga en menos tiempo o con menos nodos).

En cuanto los árboles de búsqueda, este proyecto aporta la implementación, en el lenguaje de programación Go, del pseudocódigo de la poda alfa-beta. Esta implementación también podría ser aplicada a otro proyecto, simplemente utilizando el mismo código.

7.2 Paralelización

Ligeramente relacionado con el punto anterior, otra de las posibles mejoras que se le podría implementar a este sistema, sería la paralelización de alguna de sus partes. Especialmente, tras varios análisis, se ha observado que la parte lenta se corresponde con las llamadas del algoritmo de búsqueda a la función de evaluación. Por tanto, se podría separar el árbol de búsqueda a profundidad 0, y ejecutar por separado cada uno de los posibles movimientos, y más adelante, quedarse con el que tenga mayor valor.

Esto daría como resultado, que el sistema tardase en calcular un movimiento, tanto como tardase el cálculo del movimiento más lento, lo cual sería considerablemente más rápido que el estado actual en el que se tiene que esperar a que termine el cálculo de cada movimiento para empezar el siguiente. Además esta nueva implementación llevaría cierto tiempo de adaptación, pero no requeriría hacer cambios profundos en la arquitecturas del sistema, puesto que este ya está preparado para soportar paralelización. Concretamente, en el main, divide mediante un for los posibles movimientos a nivel 0, para después calcularlos uno a uno. Si en ese punto, el cálculo se dividiese en procesos ligeros, se obtendría una gran mejora de rendimiento.

7.3 Función de evaluación

Otro de los puntos clave de este proyecto, es la conexión al motor de ajedrez de Stockfish para utilizar su función de evaluación. Una de las posibles mejoras o futuros de desarrollos en torno a este punto, sería la posibilidad de importar cualquier otro motor de ajedrez, que sería realmente sencillo de hacer, ya que sólo habría que cambiar el ejecutable que utiliza y ver si el tratamiento de la respuesta del motor de ajedrez se hace correcto. Poder ejecutar otra función de evaluación, nos permitiría hacer el mismo estudio que se ha hecho con los algoritmos de búsqueda, para determinar qué función de evaluación en más correcta en caso, por ejemplo, viendo en qué fase del juego obtiene más ventaja. Además, en torno a ese nuevo estudio y junto con este proyecto, se podría llegar a pensar en abordar el ambicioso proyecto de crear una función de evaluación desde cero, diferente a las ya existentes para tratar de batirlas.

Otra mejora posible para este proyecto, sería utilizar el código ya desarrollado para modificarlo ligeramente y darle otras funcionalidades, como por ejemplo, realizar un analizador de partidas de ajedrez, ya que, ahora mismo, este código puede servir para entrenarse, pero si se adaptase su funcionalidad, incluso podría llegar a destacar en qué puntos del entrenamiento se debe mejorar. También podría pasársele partidas ya jugadas, y que el programa fuese indicando qué ventaja existía por parte de cada jugador en cada momento, para así poder refinar el estilo del usuario del sistema.

7.4 Otra funcionalidad para el ajedrez

Al afrontar este proyecto, se barajaron distintas opciones de la funcionalidad final que podía tener el sistema. Finalmente, la decisión fue crear un jugador de ajedrez. Sin embargo, otra de las posibilidades con bastante peso, y que podría ser una futura línea de desarrollo a partir de este proyecto, es la funcionalidad de analizar partidas ya jugadas.

Este código podría adaptarse sin apenas modificaciones, para que dado una posición disponible por un jugador, analizase la jugada que este hizo, a través de la función de evaluación, y también, analizase la mejor jugada que considera el motor de ajedrez, y en base a eso, ver la diferencia entre la jugada escogida por el jugador humano y la escogida por la máquina. Esto podría ayudar a un jugador a mejorar su forma de jugar y ver los errores cometidos.

7.5 Otros juegos

Otra de las posibles líneas de desarrollo futuro de este trabajo puede ser la aplicación de las mismas técnicas utilizadas para resolver el problema del ajedrez en otros juegos con el mismo dominio del problema. Existen numerosos juegos de estrategia que son sensibles a ser resueltos con las mismas técnicas utilizadas y explicadas en este proyecto, por ello, se puede utilizar la documentación de este proyecto para tratar de extrapolar ideas a otros juegos.

Hoy en día, existe un juego de estrategia llamado Go [Schraudolph, 1994], que por distintas razones no fue conquistado por la IA hasta marzo de 2016 [Deep Mind, 2016]. Hasta esta fecha, no existía ninguna máquina capaz de ganar al considerado actualmente mejor jugador del mundo de Go, Lee Se-dol. En marzo, sin embargo, AlphaGo, máquina desarrollada por Google, venció a Lee Se-dol cuatro de las cinco partidas en las que se enfrentaron. Aun así, este juego podría ser uno de los objetivos de un posible avance de este proyecto, ya que a diferencia del ajedrez, no existe un motor de Go que sea claramente superior a los mejores jugadores humanos. Conseguir desarrollar una máquina capaz de ganar al Go, es decir, tener un objetivo a la hora de desarrollar una IA, puede hacer que sea más fácil este reto y que los problemas se vean de forma más clara, llegando a desarrollar técnicas de IA que se puedan aplicar en casi cualquier otro campo.

Al final, como puede verse con múltiples ejemplos dentro de la comunidad científica, con proyectos como el aquí detallado o con otros mucho más elaborados y complejos, científicos del todo el mundo consiguen que la ciencia avance jugando a ponerse retos complejos, es decir, de la forma más divertida posible… Siempre jugando.

Anexos

Anexo A - Presupuesto

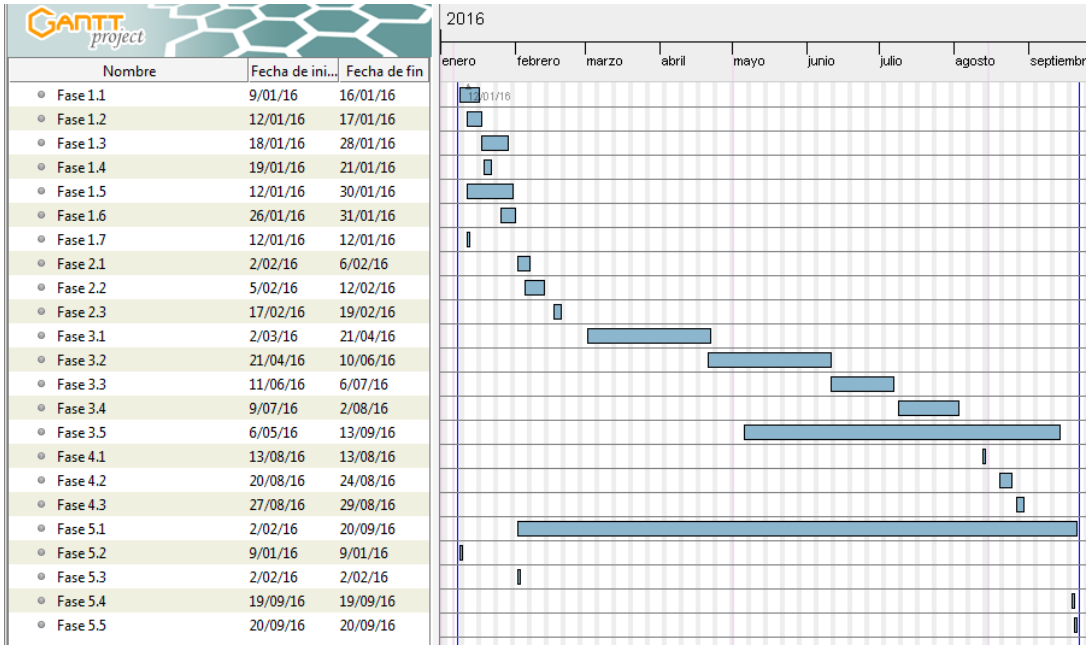
En este anexo, se explica la planificación del proyecto realizada antes del comienzo de este. En esta, se detallan las fases y los costes de personal y material asociados a la realización del proyecto.

Planificación	
Etapa 1	Análisis
Fase 1.1	Estudio de la IA aplicada al ajedrez
Fase 1.2	Estudio de los elementos de IA (funciones de evaluación, motores de ajedrez, algoritmos de búsqueda)
Fase 1.3	Análisis de mejoras al sistema (aperturas)
Fase 1.4	Análisis de los lenguajes disponibles para implementar el sistema
Fase 1.5	Extracción de requisitos
Fase 1.6	Toma de decisiones en base a los detalles analizados
Fase 1.7	Realización planificación del proyecto
Etapa 2	Diseño
Fase 2.1	Diseño del flujo de ejecución
Fase 2.2	Diseño de las estructuras de memoria
Fase 2.3	Diseño de las pruebas
Etapa 3	Implementación
Fase 3.1	Implementación del sistema capaz de seguir las reglas del ajedrez
Fase 3.2	Implementación de la conexión al motor de ajedrez para utilizar su función de evaluación

Fase 3.3	Implementación del algoritmo de búsqueda
Fase 3.4	Implementación de las estructuras para almacenar las aperturas
Fase 3.5	Revisión y mejora del código implementado
Etapa 4	Pruebas
Fase 4.1	Planificación de las pruebas a realizar
Fase 4.2	Realización de las pruebas planificadas
Fase 4.3	Evaluación de los resultados de las pruebas
Etapa 5	Documentación
Fase 5.1	Realización del documento actual
Fase 5.2	Documentación de las pruebas
Fase 5.3	Documentación de la planificación
Fase 5.4	Revisión del contenido del documento
Fase 5.5	Revisión ortográfica del documento

Tabla 46. Fases del proyecto

A continuación se muestra la gráfica de Gantt de la planificación del proyecto, detallando en cada fase, fecha de comienzo y de fin de cada tarea, así como la duración de cada una de ellas.



Nombre	Fecha de ini...	Fecha de fin
Fase 1.1	9/01/16	16/01/16
Fase 1.2	12/01/16	17/01/16
Fase 1.3	18/01/16	28/01/16
Fase 1.4	19/01/16	21/01/16
Fase 1.5	12/01/16	30/01/16
Fase 1.6	26/01/16	31/01/16
Fase 1.7	12/01/16	12/01/16
Fase 2.1	2/02/16	6/02/16
Fase 2.2	5/02/16	12/02/16
Fase 2.3	17/02/16	19/02/16
Fase 3.1	2/03/16	21/04/16
Fase 3.2	21/04/16	10/06/16
Fase 3.3	11/06/16	6/07/16
Fase 3.4	9/07/16	2/08/16
Fase 3.5	6/05/16	13/09/16
Fase 4.1	13/08/16	13/08/16
Fase 4.2	20/08/16	24/08/16
Fase 4.3	27/08/16	29/08/16
Fase 5.1	2/02/16	20/09/16
Fase 5.2	9/01/16	9/01/16
Fase 5.3	2/02/16	2/02/16
Fase 5.4	19/09/16	19/09/16
Fase 5.5	20/09/16	20/09/16

Ilustración 40. Gráfico Gantt del proyecto

Como resultado de esa planificación, se obtiene que el proyecto tiene una duración estimada de 254 días, suponiendo que, en media, se trabajan 5 horas por día, nos sale un total de 1270 horas.

El presupuesto asociado a esta planificación realizada es:

Presupuesto				
Recurso	Coste	Vida útil estimada	Uso estimado	Coste final
		Hardware		
Portátil para implementar sistema	640€	240 meses	254 días	22,57€
		Software		
Linux Ubuntu 15	0€	-	-	-
Stockfish 4	0€	-	-	-
Go 1.6.3	0€	-	-	-
Aperturas	0€	-	-	-
		Personal		
Analista, arquitecto, programador y encargado de documentación	27.375€ (5 horas/días, 15€/hora)	12 meses	254 días	19.050€
		Material fungible		
Material de oficina	15€	-	-	15€
Total				19.087,57€

Tabla 47. Presupuesto estimado del proyecto

El coste estimado total es de diecinueve mil ochenta y siete euros con cincuenta y siete céntimos de euros.

Finalmente, el coste total se vio ligeramente incrementado ya que el proyecto se finalizó con tres días de demora, pasando del 20 de septiembre al 23 de septiembre. El presupuesto real es:

Presupuesto				
Recurso	Coste	Vida útil estimada	Uso estimado	Coste final
		Hardware		
Portátil para implementar sistema	640€	240 meses	257 días	22,84€
		Software		
Linux Ubuntu 15	0€	-	-	-
Stockfish 4	0€	-	-	-
Go 1.6.3	0€	-	-	-
Aperturas	0€	-	-	-
		Personal		
Analista, arquitecto, programador y encargado de documentación	27.375€ (5 horas/días, 15€/hora)	12 meses	257 días	19.275€
		Material fungible		
Material de oficina	15€	-	-	15€
Total				19.312,84€

Tabla 48. Coste real del proyecto

El coste real del proyecto es de diecinueve mil trescientos doce euros con ochenta y cuatro céntimos de euro.

Anexo B - Project Outline

When computers solve the initial functionalities which they were created (reduction of computational times, data storages, telecommunications, …), it starts to open new investigation lines, which have as objective that these devices solve problems, each time more complex, by themselves.

Improvements in the technics capabilities of the computers (hardware innovations), and the new software that was created, it enabled to manage a major data volume much faster. All of that allow that the way in which everybody sees computer science changes. In that moment, computer science leave to be a science only used to solve engineering problems to become a science which can solve problems in any discipline.

At this point, the idea of trying to develop software which can solve a problem or to make a decision by itself was born. Improvements, which were explained before, help these programs to do more buffer operations per second, in order to increase the computational speed.

In this context, the Artificial Intelligence (AI) reaches a good level of development, which tries to simulate the human reasoning and apprenticeship in a computer. However, this AI could be thought in many different ways. It is possible to try that a computer “reasons” like a human, but with the “increased” characteristics of its hardware (more computations capabilities, processing, storing of data, …), in this way to understand the AI we find evolutionary algorithms, neural networks or humanoid robotics. On the other hand, we have

a very different way to understand the AI, if we realize that it is really difficult to simulate the behaviour of the human brain. The main problem to simulate the human brain is its complexity and that, nowadays, we do not know how it works exactly. Therefore, it is possible to develop an AI which only uses the capabilities in which a computer is better than a human. And the programmer is who defines certain rules to guide the AI. For example, with heuristics it is the human who decides what is intelligent or not, that is to say, human says to the program the pattern to reach the objective, so that it takes advantage of the capacity of the hardware to evaluate decisions to a bigger depth.

Besides, the development of the AI has many technical challenges, because it needs to do a lot of operations which consume many resources. This causes that it does not exist only one way to face the development of a program which implements AI, and it is really difficult to determine if this program is efficient or not. Thus, there are a lot of strategies to develop an algorithm, and each strategy is better depending on the problem it faces.

Nowadays, the AI is used to many different areas like robotic (in which AI tries to give more autonomy to do a not specific and repetitive task), medicine (intelligent monitoring of patients, to detect clinic pattern in biomedical signals …), economy (to analyse business failure, to estimate risks, cost effectiveness of financial products …), industrial engineering (to predictive maintenance of mechanical components, to design of vertical breakwaters …), and of course, in computer science, where evolutionary algorithms, neural networks or heuristic search apply. Also, it is possible to use AI to play games. It is a good way to try to find the most efficient way to solve it. For example, to play games with a finite

and discrete domain, like N-puzzle or Rush Hour; to strategy games, like Risk, Chess or Go; to play videogames, like shooters, graphic adventures, role games …

This project focuses on the implementation of a program which plays chess. The program uses an evaluation function and a search tree to find the best movement in each play in the less time as possible.

Chess is a strategy game in which two players play one against the other. Players have different pieces to move them on a board. The chess kit is composed of sixteen chessmen which can be moved with restrictions across a board with sixty-four boxes, for this reason, the possible combinations are practically infinite. The origin of the chess is unknown, but it is possible to affirm that is really old because, in excavations in Mesopotamia, objects were found which show that chess or similar games already exited four thousand years before Christ. So, both his history and its game structure, chess is a much known game and played in whole world. Chess is also a good game to play it using every known AI technics. In 1996, IBM developed a supercomputer called Deep Blue which managed to win the world champion that year, Gary Kasparov.

The goal of this project is to create a chess player which do not learn movements, only follow rules, since the beginning of the game. Chess players follow a heuristic which is defined earlier.

Besides, in the documentation, it is details how works an evaluation function applied to games like chess. Also, it is explained how works the different search trees which can be used with this heuristics. Afterwards, it is explained some of the most important aspects of chess that are necessary to do this project.

This project has different purposes which can be described as:

- One objective is to implement a software in the programming language Go, which has the capacity to follow the rules of the International Federation of Chess. Besides, is mandatory that this software represents any chess state and generates the possible movements for each piece.
- In order to decide which movement is better to be executed, the system is going to implement a function to connect it with the chess engine, Stockfish. This connection will be done through UCI interface, and the system only is going to use the evaluation function of the Stockfish, not the other functionalities that it offers.
- When it exists the possibility of execute an evaluation function, is necessary to implement a search algorithm which can found the best movement in a big depth. To reach this objective, the system is going to use a search algorithm called alfa-beta pruning. The algorithm executes movements in different depths and, when it find a leave, it calls to the evaluation function and return the movement with the best value, according to the evaluation function of Stockfish.
- Once that the system has been finished is necessary, first, check if everything works properly; second, execute the system against another chess engine to analyse the results that both systems have. In order to know which player is better, three parameters will be analysed, expanded nodes, time spent and the final result of the game.

- In order to help the system to works faster, it exits the objective of develop a functionality to ingest an encyclopaedia of chess openings as much large as possible. This functionality will help the system because it avoids calculating the best movement, it only follows the movement written in the list of openings.

Once objectives are defined, it is important to explain how the system has been built. First of all, the structures to represent the board and the possible movements from every state are the following:

- The variable called “board”, it is an array of 64 positions of integers which has the state of each square in each position of the array. The possible value stored in the positions of the array are:
    - Positive numbers if there is a white chess piece. 71 for rooks, 72 for knights, 73 for bishops, 74 for queen, 75 for king and 76 for pawns.
    - Negative number if there is a black chess piece. -71 for rooks, -72 for knights, -73 for bishops, -74 for queen, 75 for king and -76 for pawns.
    - 0 if this square is empty.

- The variables called “movsLists” which are arrays of lists. Each variable is an array of 64 positions which contains in each position of the array a list with all possible movements of the chess piece of the array. This

variable is very useful because it avoids recalculating all possible movements for one piece in each movement.

- The variable called “openingList” which is also an array of lists. Each position of the array contains a list of all movements of one variation of an opening. It is means that if the system wants to get one movement of one opening, it only need to access to the position of this opening a get the next movement.

The combination of these variables, allows executing one movement. Steps to change the state of the board are the following.

First, it is necessary to read the state of the board. This is done in the function main which has always the actual state of the board. Then, with a loop, the system gets the state in each square of the board, when it is find a square with a chess piece, it is calls another function passing as parameters the square of the board and the chess piece in it.

Afterwards, the function which calculates movements receives the square and the chess piece and, according to the piece, it gets all moves of it. Once that it has the possible movements, the function cuts these movements to get only the legal movements in actual state of the square.

Once the system has all legal movements in the square, it calls to the function to execute the search tree. It receives a state of the board, the player who is the next to execute a movement and the depth to search the solution. Then, it starts its execution. It does many recursive calls until it find a leave, in this case, it

invokes the evaluation function. Afterwards, and following the alfa-beta pruning, the search tree promotes better values until it reaches a state in which all states have been evaluate. Then, the function of the search tree returns the best movement according to the evaluation function of the Stockfish.

However, there is another possibility of execution, instead of using the evaluation function and the search tree, it is possible to use directly the encyclopedia of chess openings.

In the case that the system uses openings, the first step that is done it is to choose the opening that is going to be executed. To choose the opening, it is necessary to go down of all positions of the array. Each position has in the front of the list a number (0 or 1) which indicates if the opening is available according to the movements which have been done. If there is a 0 stored in the front means that the opening is not available, and, on the other hand, if it is stored a 1 it means that the opening is available. At the end, the opening chosen will be the longest opening among the available openings.

Besides, when the opponent executes a movement, it is also necessary register it. To do this, the system reads the movement and it calls to a function which searches if the movement is included in the openings available. If the movement is included in the list of movements of the opening, it will not happen anything. Nevertheless, if the movement is not included, this opening will be eliminated of the list storing a 0 in the front of its list.

Once the system is finished, the main results are summarized, evaluating if they fit the model which was initially raised. The problems founded during the fulfillment of this project will be mentioned as well, and the results and conclusions obtained at the end of this project will be highlighted.

Below are listed the objectives set initially, evaluating if they have been achieved or not and explaining the reasons.

- **Software in Go:** In this section the objective which was set was to get implement software capable of playing chess according to international rules. This objective has been met, since the system has the ability to calculate all the legal movements available in all possible situations. In addition, it was stipulated that this had to have been done in the Go programming language, which is precisely the only language used throughout the system.

- **Function Evaluation:** This target indicated that the system should be able to connect to the function evaluation chess engine Stockfish. This has been achieved by making a call to Stockfish by UCI interface, and processing the response for the value of the evaluation function.

- **Search algorithm:** This target indicated that the system should implement a search algorithm to find the best move using the evaluation function of Stockfish. This algorithm, alfa-beta pruning, should be executed, at least,

with 4 depth. This objective has been reached because the system uses an alfa-beta pruning to do a dynamically strategy during the game in the depth that player whish. However, and although the objective does not speaks about time, the purpose was that the system would execute movements in a reasonable time. Nevertheless, the functionality of searching movements in a big depth is quite slow, for this reason, the system is not able to play a game against a human player, because the game would not be enough fast to beat the human.

- **Analysis of algorithms:** In this section the objective which was set was to analyze the search algorithm of the system, and to extract information from it related with the performance of the algorithm. This objective has been reaches as it is written in the test chapter.

- **Openings:** The target of this section was to build structures in memory to store a large encyclopedia of chess openings. Besides, it was necessary to implement functions to allow the system to access to the openings. This objective has been reached because the system has access to a file with 2.281.761. It also has functions to store these openings in memory and, in this way, the open are able to be accessed during the execution. Openings have been stored at the beginning of the execution of the system, in order to avoid an overload in the system during the normal execution of it.

It is possible to see that all objectives have been reached. In this point that the documentation of the system has been finished, it is conveniently to explain which the futures works to continue the project are.

One of the main improvements which should be designed it is a new search algorithm which improve the performance of the system. This new algorithm can be the implementation of an existing algorithm or a new implementation of a search algorithm. This project allows to know how works a chess engine and search algorithm and how they are related, so using it as an initial point, it possible to develop an algorithm which solves some of the problems of AI nowadays.

Another future work, related with this project, can be the implementation of different functionalities of the program. For example, now, the system is used to play chess, but, with a few changes, the system can be used an analyzer of chess game. So, a player can use this new program to learn how to play chess better.

The last future work, related with this project, is the option to use the same techniques that are used in the system to play a different game. So, it is possible to use the same theory to play other games like Go game. Then, the project provides the theory to play to any game, in a heuristic way.

Anexo C - Bibliografía

[CITIC, 2014] Centro de Investigación en Tecnologías de la Información y de la Comunicación (CITIC) [sede web]. A Coruña: Universidad de A Coruña. 2014. [acceso 25 de julio 2016]. Inteligencia Artificial y Aplicaciones. Disponible en: http://citic-research.org/area_tecnologica/2?locale=es

[Fénix, 2012] Fénix J., Ajedrez: características, curiosidades y beneficios. [monografía en Internet] Argentina: Sentido común 3.14; 2012. [acceso 25 de julio 2016]. Disponible en: http://sentido314.blogspot.com.es/2012/01/ajedrez-caracteristicas-curiosidades-y.html

[Campbell, 2002]Campbell M.,Hoane Jr J.A., Hsuc F. Deep Blue. Artificial Intelligence[revista en Internet] 2002 [acceso 26 de julio 2016]; 134 (57-83). Disponible en: http://www.sciencedirect.com/science/article/pii/S0004370201001291

[Parra, 2013]Parra S. Las cifras más alucinantes del ajedrez.[monografía en Internet] España: xataka ciencia; 2013. [acceso 26 de julio 2016]. Disponible en: http://www.xatakaciencia.com/matematicas/las-cifras-mas-alucinantes-del-ajedrez

[Wikipedia, 2016]Wikipedia [sede web]. 2016 [acceso 11 de agosto 2016]. Elo rating system. Disponible en: https://en.wikipedia.org/wiki/Elo_rating_system

[FIDE, 2015]World Chess Federation (FIDE) [sede web]. 2015 [acceso 12 de agosto de 2016]. Disponible en: https://www.fide.com/

[FIDE, 2015]World Chess Federation (FIDE) [sede web]. 2015 [acceso 12 de agosto de 2016]. Standard Top 10 Players September 2016. Disponible en: https://www.fide.com/https://ratings.fide.com/top.phtml?list=men

[CCRL, 2016]Computer Chess Ranking List (CCRL) [sede web]: CCRL 40/40. 2005 - 2013 [actualizada el 17 de septiembre de 2016; acceso 19 de septiembre de 2016]. Downloads and Statistics. Disponible en: http://www.computerchess.org.uk/ccrl/4040/

[WBEC,2001]-ridderkerk [sede web]. Ridderkerk: Netherlands. 2001. [actualizada el 28 de abril de 2004; acceso 20 de agosto 2016]. Description of the universal chess interface (UCI). Disponible en: http://wbec-ridderkerk.nl/html/UCIProtocol.html

[CPW, 2016] Chess Programming Wiki (CPW) [sede web]. CPW. 2016 [acceso 20 de agosto de 2016]. Universal Chess Interface (UCI). Disponible en: https://chessprogramming.wikispaces.com/UCI

[Mann, 2003] Mann T., Chess Engine Communication Protocol [Internet] California: jChess; 2003 [acceso 20 de agosto de 2016]. Disponible en: http://jchecs.free.fr/pdf/XBoardProtocol.pdf

[CPW, 2016] Chess Programming Wiki (CPW) [sede web]. CPW. 2016 [acceso 25 de agosto de 2016]. Forsyth-Edwards Notation (FEN) Disponible en: https://chessprogramming.wikispaces.com/Forsyth-Edwards+Notation

[CPW, 2016] Chess Programming Wiki (CPW) [sede web]. CPW. 2016. [acceso 25 de agosto de 2016]. Komodo. Disponible en: https://chessprogramming.wikispaces.com/Komodo

[Hartmann, 2013] Chess News [sede web]. Hamburg: Germany. 2013. [actualizada el 28 de septiembre de 2014; acceso 30 de agosto 2016]. Hartmann – choosing a chess engine. Disponible en: https://en.chessbase.com/post/hartmann-choosing-a-chess-engine

[Quora, 2015] Anonymous. What is the algorithm behind Stockfish, the chess engine?[monografía en Internet] Quora; 2015. [acceso 30 de agosto 2016]. Disponible en: https://www.quora.com/What-is-the-algorithm-behind-Stockfish-the-chess-engine

[CPW, 2016] Chess Programming Wiki (CPW) [sede web] CPW. 2016 [acceso 30 de agosto de 2016]. Stockfish. Disponible en: https://chessprogramming.wikispaces.com/Stockfish

[Lagrain, 2012] Chess News [sede web]. Hamburg: Germany. 2013. [actualizada el 29 de octubre de 2012; acceso 30 de agosto 2016]. Jan Lagrain - Houdini 3- the world's strongest chess engine in the Fritz interface. Disponible en: http://en.chessbase.com/post/houdini-3-the-world-s-strongest-che-engine-in-the-fritz-interface

[Sion, 1958] Sion M. On general minimax theorems. In: Gilbarg D., Beaumont R.A., Whieteman A.L., Straus E.G., editors. Pacific Journal of Mathematics. 1st ed. Unites Estates of America (USA). 1958. p. 171–176. Disponible en: http://msp.org/pjm/1958/8-1/pjm-v8-n1-s.pdf#page=173

[Fan, 1952] Ky Fan. Minimax Therorems. University of Notre Dame and American University. [17 de noviembre de 1952, acceso 30 de agosto 2016]. Disponible en: http://www.pnas.org/content/39/1/42.short

[CPW, 2016] Chess Programming Wiki (CPW) [sede web]. CPW. 2016 [acceso 30 de agosto de 2016]. Horizon Effect. Disponible en: https://chessprogramming.wikispaces.com/Horizon+Effect

[Berliner, 1973] Berliner B. Some necessary conditions for a master chess program. Carnegie-Mellon University. Pittsburgh, Pennsylvania. [1973]. Disponible en: http://ijcai.org/Proceedings/73/Papers/010.pdf

[UIB, 2016] Introducción a la Inteligencia Artificial [sede web]. España: Universidad de las Islas Baleares. 2016 [acceso 30 de agosto de 2016]. Espera del reposo. Disponible en: http://dmi.uib.es/~abasolo/intart/2-juegos.html#2.3.3.2

[Fuller, 1973] Fuller S.H., Gaschnig J.G., Gillogly, Analysis of the alpha-beta pruning algorithm. [Internet]. Pittsburgh, Pennsylvania : Department of Computer Science Carnegie-Mellon University; 1973 [acceso 30 de agosto de 2016]. Disponible en: http://repository.cmu.edu/cgi/viewcontent.cgi?article=2700&context=compsci

[Wikipedia, 2016]. Wikipedia [sede web]. 2016 [actualizada 15 de junio de 2016; acceso 31 de agosto de 2016]. Poda alfa – beta. Disponible en: https://es.wikipedia.org/wiki/Poda_alfa-beta

[CPW, 2016] Chess Programming Wiki (CPW) [sede web]. CPW. 2016 [acceso 1 de septiembre de 2016]. Alpha - Beta. Disponible en: https://chessprogramming.wikispaces.com/Alpha-Beta

[CPW, 2016] Chess Programming Wiki (CPW) [sede web]. CPW. 2016 [acceso 1 de septiembre de 2016]. Quiescence Search. Disponible en: https://chessprogramming.wikispaces.com/Quiescence+Search

[Griesemer, 2009] Griesemer r., Pike R., Thompson K. The Go Programming Language [sede web]. USA .Go. 2009. [acceso 5 de septiembre de 2016]. Disponible en: https://golang.org/

[Griesemer, 2009] Griesemer r., Pike R., Thompson K. The Go Programming Language [sede web]. USA . Go. 2009. [acceso 5 de septiembre de 2016]. What is the history of the project? Disponible en: https://golang.org/doc/faq#history

[Pike, 2012] Pike R. Go at Google: Language Design in the Service of Software Engineering [monografía en Internet] USA: The Go Programming Language; 2012. [acceso 5 de septiembre de 2016]. Disponible en: https://talks.golang.org/2012/splash.article

[Benchmarks, 2014] The computer Language Benchmarks [sede web]. The computer Language Benchmarks; 2014 [acceso 5 de septiembre de 2016]. Go programs versus Java. Disponible en: https://benchmarksgame.alioth.debian.org/u64q/go.html

[PGN Mentor, 2008] PGN Mentor [sede web]. PGN Mentor; 2008 [acceso 10 de septiembre de 2016]. Players. Disponible en: http://www.pgnmentor.com/files.html#players

[Schraudolph, 1994] Schraudolph N.N., Dayan P., Sejnowski T.J. Temporal Difference Learning of Position Evaluation in the Game of Go. [Internet]. Computational Neurobiology Laboratory The Salk Institute for Biological Studies. California; 1994 [acceso 5 de septiembre de 2016]. http://www.variational-bayes.org/~dayan/papers/sds94.pdf

[Deep Mind, 2016] Deep Mind [sede web]. USA. Deep Mind; 2016 [acceso 5 de septiembre de 2016]. Alpha – go. Disponible en: https://deepmind.com/alpha-go